November 2007

Principia Narcissus: How to Avoid Being Caught by Your Reflection

Geoffrey Alan Washburn
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-07-25.

Principia Narcissus: How to Avoid Being Caught by Your Reflection

Abstract

Some modern, statically typed programming languages provide the capability for programs to reflect, or introspect, upon their type meta-data at runtime. Using type meta-data to determine program behavior is called *type-directed programming (TDP)*. Type-directed programming allows many operations on data, such as serialization, cloning, structural equality, and general iteration, to be defined naturally, just once, for all types of data. Consequently, these operations continue to work as systems grow and software is extended with additional data types. Without TDP, programmers must constantly revise the code that implements these operations and scatter their implementations throughout their code-base.

However, TDP conflicts with the use of abstract data types (ADTs), a fundamental technique in the practice of software engineering. The benefits of using ADTs derive from the fact that their definitions are hidden; however, with TDP, abstract type meta-data becomes no more hidden than abstracted values (often called variables) in standard programming.

In this dissertation, I show how TDP and ADTs can be reconciled through the use of information-flow type and kind systems. I begin by introducing the problem as well as my definitions for the properties I call confidentiality and integrity. Next, I develop the theoretical foundation for reasoning statically about confidentiality and integrity in programs that use TDP, and show how information-flow type and kind systems generalize prior techniques. I then describe a realistic programming language, InforML, with an information-flow type and kind system. After introducing the InforML language, I describe idioms for programming in InforML and the reasoning principles for confidentiality and integrity that are a consequence of using these idioms. Finally, I discuss the implementation of InforML and the most important design decisions made while implementing InforML.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-07-25.

PRINCIPIA NARCISSUS: HOW TO AVOID BEING CAUGHT BY YOUR REFLECTION

Geoffrey Alan Washburn

A DISSERTATION

in Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2007

Technical Report MS-CIS-07-25

For Mom and Dad.

ABSTRACT

PRINCIPIA NARCISSUS: HOW TO AVOID BEING CAUGHT BY YOUR REFLECTION

Geoffrey Alan Washburn

Stephanie Claudene Weirich

Some modern, statically typed programming languages provide the capability for programs to reflect, or introspect, upon their type meta-data at runtime. Using type meta-data to determine program behavior is called *type-directed programming* (TDP). Type-directed programming allows many operations on data, such as serialization, cloning, structural equality, and general iteration, to be defined naturally, just once, for all types of data. Consequently, these operations continue to work as systems grow and software is extended with additional data types. Without TDP, programmers must constantly revise the code that implements these operations and scatter their implementations throughout their code-base.

However, TDP conflicts with the use of abstract data types (ADTs), a fundamental technique in the practice of software engineering. The benefits of using ADTs derive from the fact that their definitions are hidden; however, with TDP, abstract type meta-data becomes no more hidden than abstracted values (often called variables) in standard programming.

In this dissertation, I show how TDP and ADTs can be reconciled through the use of information-flow type and kind systems. I begin by introducing the problem as well as my definitions for the properties I call confidentiality and integrity. Next, I develop the theoretical foundation for reasoning statically about confidentiality and integrity in programs that use TDP, and show how information-flow type and kind systems generalize prior techniques. I then describe a realistic programming language, InforML, with an information-flow type and kind system. After introducing the InforML language, I describe idioms for programming in InforML and the reasoning principles for confidentiality and integrity that are a consequence of using these idioms. Finally, I discuss the implementation of InforML and the most important design decisions made while implementing InforML.

Contents

List of Tables

List of Figures

Preface

The beginning is the most important part of the work.
Plato (*The Republic*)

Gratitude is not only the greatest of virtues, but the parent of all the others.
Cicero (*Pro Plancio*)

I first began thinking about the problem of reconciling type-directed programming and representation independence in May of 2003, not long after my advisor, Stephanie Weirich, and I published our first paper together. It was one of several research problems surrounding intensional type analysis we discussed.

The first time I recall having thought about recovering representation independence using information-flow kind and type systems was at the end of June 2003, at the University of Oregon Foundations of Security Summer School. I remember asking Steve Zdancewic, after his lecture on information flow, whether the idea made any sense.

The next time I remember thinking about information-flow kind and type systems was early January 2004, when I was invited to give a talk as part of the IFIP Working Group 2.3. I had been invited to be a student participant, in exchange for handling some of the organizational tasks. Chiefly, I recall being tasked with securing cheese-steaks from John's Roast Pork in South Philly, for an outing at the Constitution Center. I had considered giving a very rough presentation on my idea of using information-flow kind and type systems, but, in the end, I decided that without having spent any time working out the details, it would be best to decline the offer.

However, near the end of February 2004, I began working on a ICFP submission that would prove a version of parametricity using an information-flow kind and type system. At that time, the paper had the rather punny title "Cloak and Dagger: Type-Directed Programming with Information Flow" – I was using dagger (†, ‡) and "bag" notation as part of the language's meta-variables and syntax. I do not remember whether it was the title or the notation that came first. However, about twenty-four hours before the deadline, Stephanie and I decided that the paper was not going to be polished enough to be a respectable submission.

Stephanie and I resumed work on the paper with the aim of submitting to POPL; the paper now had the more staid title "Generalizing Parametricity Using Information Flow". In the end, we did get the paper together and submitted it to POPL in July of 2004. Considerable thanks goes to Simon Peyton Jones who allowed me to devote time to this paper, even though the primary goal of my internship at Microsoft Research Cambridge was to work on generalized algebraic data types. This paper was not accepted into

the conference programme, but we received several helpful reviews that allowed for us to improve the presentation of the ideas.

Fairly late in 2004, I began to work on revising the paper but wound up taking a two detours. I was intrigued by Eijiro Sumii's work on using bisimulations to prove that abstract data types, including those that contain recursive types, were contextually equivalent. The former did not really lead to any interesting results, because constructing a bisimulation will only tell you about the relationship between two specific abstract data types, while it is possible to derive properties about any abstract data type with a certain signature using generalized parametricity.

Also in late in 2004, I tried working out an alternative proof of generalized parametricity that used an effect system, rather than an information-flow kind and type system. Trying to use a language with an effect system in the proof failed, but it helped me better understand that the way information-flow kind and type systems make dependencies explicit is key to making the proof of generalized parametricity work out.

Early in 2005, I completed my revisions to "Generalizing Parametricity Using Information Flow" and submitted the paper LICS. This version of the paper was accepted into the program, and formed the theoretical basis for all of future work on reconciling type-directed programming and reasoning about data abstraction. It was the seed from which this dissertation crystallized.

Despite having my name of the title page, this dissertation owes its existence to so many others. I cannot hope to properly thank each of those individuals here, but I will try my best. If there is someone I have forgotten, please forgive me – there are so many of you to remember.

Firstly, this dissertation would not have been possible without my parents, George and Sharon Washburn. Aside from their obvious contribution to my own existence, throughout my life they have given me considerable love and support in its many forms. From signing me up for classes on Logo programming, when I was so young that I cannot even remember how old I was at the time, to helping pay for the vast majority of my undergraduate education at Carnegie Mellon, they have helped me to get to where I am today. Finally, much of this dissertation was written while I stayed with them, after I allowed my lease on my apartment in Philadelphia to expire.

My advisor, Stephanie Weirich, as I described above, suggested the research problem this dissertation solves and helped with the development of generalized parametricity. However, those two contributions are only a small fraction of the ways that she has helped me. During my time at Penn, she has done everything from introducing me the Siamese breed of cat to working hard reading drafts of this dissertation to provide me with critical feedback. I cannot imagine having had a better advisor than Stephanie, and I am truly lucky to have worked with her.

Steve Zdancewic deserves nearly as much credit as Stephanie. Despite the fact that we have only done a little research together, I have easily spent as much time with him while at Penn as Stephanie. Because Steve's research has often directly involved information-flow type systems, he has been an invaluable resource throughout the research behind this dissertation. I am also grateful that he agreed to serve as the chair of my thesis committee.

Benjamin Pierce, foremost deserves my thanks for simply being at Penn. Without his presence, research on types and programming languages at Penn would not be flourishing today. Even though I spent only a single semester while at Penn doing research with Benjamin, he has always been valuable resource on matters professional, personal, and artistic. He also has my thanks for agreeing to serve on my thesis committee.

While I was an undergrad at Carnegie Mellon, Frank Pfenning taught my first lectures on functional programming in Standard ML, and co-advised my senior thesis project with Peter Lee. Frank taught me a great deal about constructive logic and formal proofs, of both, the paper and the mechanized variety. He also set a standard for mathematical rigour that I strive to achieve.

There were several people that, among other things, made specific contributions to this dissertation. Below, I list these individuals and their contributions:

- Brian Aydemir, for doing some last minute proofreading.
- Daniel Dantas, for all his help in the development of AspectML, the precursor to InforML.
- Derek Dreyer, for pointing out a subtle flaw in the original version of the proof of generalized parametricity.
- Vesa Karvonen, for his work on the Standard ML extended basis library. It saved me from needing to reinvent the wheel quite a often during the development of InforML.
- Peng Li (李鹏), for all sorts of help and advice with both my ThinkPad X31 and with my ThinkPad X61. The vast majority of InforML was implemented on the former and the vast majority of this dissertation was written on the latter.
- Martin Odersky, for hiring me to work with him on the Scala language at EPFL. I can only guess how much longer this dissertation would have dragged out without the introduction of a deadline.
- J Proctor, for suggesting the name InforML for my language.
- Val Tannen, for agreeing to serve not only on my thesis committee, but my WPE-II committee, and for his uncompromising standards.
- Stephen Chun-to Tse (謝鎮滔), for helping me with the underspecified parts of the Office of Graduate Studies's formatting requirements.
- Aaron Turon, for his considerable assistance in getting started with `ml-ulex` and `ml-antlr`, which were used to implement the InforML frontend.
- David Walker, for numerous conversations that have made this dissertation stronger, his collaboration and support while working on AspectML, and agreeing to serve on my thesis committee.

The following individuals, as far as I can remember, did not necessarily contribute in a direct fashion to this dissertation, but should be acknowledged regardless: Ada (my cat), Bastet, Aaron Bohannon, John Bucy, Margaret DeLap, Joshua Dunfield, Nate Foster, Freyja, Vladimir Gapyeva, Bob Harper, Limin Jia, Assaf Kfoury, Christopher League, Peter Lee, Michael Levin, Michael May, Karl Mazurak, Elizabeth

McCullough, Alexandra McLean, Nick Montfort, Simon Peyton Jones, Alan Schmitt, Chung-chieh "Ken" Shan (單中杰), Michael Smith, Mark Stehlik, Jeff Vaughn, Daniel Vogel, Dimitrios Vytinotis, Hanna Wallach, George Washburn IV, Kate Washburn, Joe Wells, and Ellie Zdancewic.

While they cannot be named, I also appreciate the feedback I received on "Generalizing Parametricity Using Information Flow" from the POPL 2005 and LICS 2005 anonymous reviewers.

Finally, some of the research that is part of this dissertation was supported by NSF grant 0347289, CAREER: Type-Directed Programming in Object-Oriented Languages.

Geoffrey Alan Washburn
November 2007
Frederick, Maryland

1

The problem

With great power comes great responsibility.
Uncle Ben (Stan Lee, *Amazing Fantasy #15*, 1962)

Some modern, statically typed programming languages provide the capability for programs to reflect, or introspect, upon their type meta-data at runtime. For example, Java (Gosling et al. 2005) and the .NET Common Language Infrastructure (ECMA 2006), the basis for many languages including C# (ECMA 2006) and F# (Syme and Margetson 2006), provide primitive operators and libraries to do so. Using type meta-data to determine program behavior is called *type-directed programming* (TDP). Type-directed programming allows many operations on data, such as serialization, cloning, structural equality, and general iteration, to be defined naturally, just once, for all types of data. Consequently, these operations continue to work as systems grow and software is extended with additional data types. Without TDP, programmers must constantly revise the code that implements these operations and scatter their implementations throughout their code-base.

However, TDP conflicts with the use of abstract data types (ADTs), a fundamental technique in the practice of software engineering (Parnas 1972). The benefits of using ADTs derive from the fact that their definitions are hidden; however, with TDP, abstract type meta-data becomes no more hidden than abstracted values (often called variables) in standard programming.

In this dissertation, I show how TDP and ADTs can be reconciled through the use of information-flow type and kind systems. I begin by introducing the problem as well as my definitions for the properties I call confidentiality and integrity. Next, I develop the theoretical foundation for reasoning statically about confidentiality and integrity in programs that use TDP, and show how information-flow type and kind systems generalize prior techniques. I then describe a realistic programming language, InforML, with an information-flow type and kind system. After introducing the InforML language, I describe idioms for programming in InforML and the reasoning principles for confidentiality and integrity that

```
data Company  = C [Dept]
data Dept     = D Name Manager [SubUnit]
data SubUnit  = PU Employee
              | DU Dept
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Float
type Manager  = Employee
type Name     = String
type Address  = String
```

Figure 1·1: Haskell data types and type definitions that describe a company (Lämmel and Peyton Jones 2003).

are a consequence of using these idioms. Finally, I discuss the implementation of InforML and the most important design decisions made while implementing InforML.

In this chapter, I begin by illustrating the utility of TDP by showing how it can be used to concisely implement traversals over arbitrary data. I then use this same example to show the tension between TDP and ADTs. Following a review of existing mechanisms that have been or could be applied to the problem, but fail to provide flexible static reasoning principles concerning data abstraction, I show how an information-flow type and kind system succeeds where these other mechanisms fail. I then conclude with an overview of the contributions found in this dissertation and a road-map for the remainder of the document.

§ 1·1 The power of type-directed programming

Type-directed programming has become recognized as an effective tool for "scrapping" the significant amount of "boilerplate" code for algebraic pattern matching that arises in modern statically typed functional languages such as ML (Milner et al. 1997; Leroy et al. 2000), F# (Syme and Margetson 2006), Scala (Odersky 2007), Clean (Brus et al. 1987), and Haskell (Peyton Jones 2003).

Imagine starting with a representation of a company written in Haskell as shown in Figure 1·1. If a programmer wants to write a function for increasing the salary of all employees in the company by a percentage (say, to adjust for inflation), it would typically be written using large amounts of tedious pattern matching code like that found in Figure 1·2.

Lämmel and Peyton Jones (2003; 2004; 2005) have shown how TDP in Haskell can be used to define a type-directed mapping function called `everywhere` that, when supplied with a function of type `forall a.a -> a`, applies it to every component of any given value. However, for `everywhere` to be useful it must be possible to write functions of type `forall a.a -> a` that are not either the identity function or divergent terms. Therefore, they also provide the lifting function `mkT`, that lifts a function of type `t -> t`, for some type `t`, to be of type `forall a.a -> a`. The resulting function is the identity

```
increase :: Float -> Company -> Company
increase p (C ds) = C (map (increaseD p) ds)

increaseD :: Float -> Department -> Department
increaseD p (D nm mgr us) =
  D nm (increaseE p mgr) (map (increaseU p) us)

increaseU :: Float -> SubUnit -> SubUnit
increaseU p (PU e) = PU (increaseE p e)
increaseU p (PD d) = PD (increaseD p d)

increaseE :: Float -> Employee -> Employee
increaseE p (E per s) = E per (increaseS s)

increaseS :: Float -> Salary -> Salary
increaseS p (S s) = S (s * (1 + p))
```

Figure 1·2: A function to increase the salary of all `Employees` in a `Company`. The function is written in terms of helper functions for the components of a `Company` (Lämmel and Peyton Jones 2003).

on data with any type other than `t`. Using `everywhere` and `mkT` it is possible to write a version of the `increase` function from Figure 1·2 succinctly:

```
increase :: Float -> Company -> Company
increase p = everywhere (mkT (increaseS p))

increaseS :: Float -> Salary -> Salary
increaseS p (S s) = S (s * (1 + p))
```

In the code above, the programmer only writes the important case, the one that increases a `Salary` value by the given percentage, and then uses `mkT` to create a function that works on any type. If the type is `Salary` then `increaseS p` is called, otherwise the identity function is used. The type-directed function `everywhere` then walks over all of the constructors and components that make up a value of type `Company` and applies the function `mkT (increaseS p)` from the bottom up. The result is that anywhere a value of type `Salary` occurs in the input `Company`, the function `increaseS p` will be called to increase the salary. Consequently, every salary in the provided `Company` will be increased.

Repetitive boilerplate code is not limited to languages with algebraic data types and pattern matching like ML and Haskell; the same problems with boilerplate arise in object-oriented languages too. In object-oriented languages, an experienced programmer might implement the same sort of traversals using the *visitor* design pattern (Gamma et al. 1994). Using the visitor pattern requires that the programmer implement cases for all classes to be traversed, even if there are only a few important cases. Furthermore, the naïve implementation of the visitor pattern, in a language like Java, has the problem that it will only work for those classes that implement a specific visitor interface. Palsberg and Jay (1998) have shown

how the Java reflection libraries can be used to implement a type-directed version of the visitor pattern that works for arbitrary Java objects.

There are many more examples of successful uses of TDP for a variety of applications:

- Java's reflection libraries have been used to provide tools for interacting graphically with components called "Beans" (Sun Microsystems 1997).
- Vestin (1997), in his masters thesis, has shown TDP can be used for implementing genetic algorithms.
- Jeuring and Hagg (2002) have shown how to use TDP, in Generic Haskell (de Wit 2002), to easily generate XML tools, such as editors and compressors.
- Jansson and Jeuring (1998) have shown how to use TDP to perform unification on arbitrary first-order data.
- Achten, et al. (2004B) have shown how to use TDP, in the Clean language, for generating gui views from arbitrary first-order data, and later with van Weelden (2004A) described how it could be extended to handle higher-order data.
- Cheney (2005) has shown how to extend the ideas developed by Lämmel and Peyton Jones to write a library so that programmers can create their own data types with binding structure, similar to the capabilities of languages like FreshOCaml (Shinwell and Pitts 2005).
- Mitchell and Runciman (2007) have developed a library for type-directed traversals that, among other applications, has been used to extensively as part of the implementation of the Yhc Haskell compiler (Golubovsky, Mitchell, and Naylor 2007).

Programmers and researchers will no doubt continue developing new and compelling applications of TDP.

However, despite all the benefits to writing software using TDP, it can make reasoning about the properties of abstract data types difficult, if not impossible. In the next section I will explain the conflict between TDP and abstract data types.

§ 1·2 Type-directed programming and abstract data types

Using abstract data types has long be recognized as a fundamental technique in the practice of software engineering (Parnas 1972). Many of the benefits of using abstract data types derive directly from the fact that the interface for an abstract data type can be *independent* from its implementation. By independent, I mean that programs written against the ADT's interface will behave the same regardless of how the ADT is implemented.[1] In statically typed languages, interfaces give names to types and operations on them

1. This is, of course, not completely true. If changing the implementation of an ADT did not change the program's behavior it is unlikely it would ever be changed. A common reason to switch ADTs is to improve performance, but reasoning at that level is beyond the scope of my dissertation. The ability to abstract away from low level details, like performance characteristics, is part of what makes reasoning with ADTs compelling.

and implementations provide definitions of those types and operations. For example, if the interface for an ADT for sets is independent of its implementation, it would be possible to implement sets either as lists of elements or as hash-tables of elements, and yet not impact the behavior of the overall program.

If a language has the property that the behavior of programs is independent of all possible ADTs, that is it is possible to freely switch between different implementations of the same interface for any given ADT, I say that the language has *representation independence*. For example, the polymorphic λ-calculus (Girard 1972; Reynolds 1974) is a language with representation independence.

I will frequently say that an ADT has the property of *confidentiality*, which means that how the interface of a *specific* ADT is implemented does not affect the behavior of some subset of a program. For example, I might say that the an ADT for sets has confidentiality inside a function for converting sets to strings. Or I might say that an ADT for complex numbers has confidentiality with respect to the entire program. If a language has representation independence, it is a corollary that every ADT has confidentiality with respect to all subsets of a program. On one hand, if every ADT in a program has confidentiality with respect to all subsets of a program, this does not imply that the language has representation independence. On the other hand, if an ADT does not have confidentiality with respect to some part of the program, that implies that the language does not have representation independence.

TDP, by definition, changes the behavior of operations based upon the type of their inputs. For example, in the previous section, if the type-directed function `everywhere` is applied to a value of type `Company` it will call itself recursively on the list argument to the `C` data constructor, and when applied to a value of type `Employee` it will call itself recursively on the `Person` and `Salary` arguments to the `E` data constructor. Because a type-directed operation can alter its behavior based on the type of its inputs, how an ADT has been implemented can affect the behavior of the program. Therefore, representation independence does not hold in a language with TDP, because ADTs do not have confidentiality with respect to type-directed functions.

To illustrate how TDP can violate the confidentiality of an ADT, consider the example data types from the previous section. Imagine if companies were defined using the same constructors as in Figure 1·1, but given an interface like the `Companies` module in Figure 1·3. The identifiers in the parenthesized block following `module Companies`, in Figure 1·3, are the exports for the module. Therefore, `Companies` exports the data types (`Company`, `Dept`, etc.) while the actual data constructors, (`C`, `D`, etc.) are hidden. Because `Companies` hides the data constructors, it must provide accessor and constructor functions for each of the data types. Additionally, the `Companies` module provides functions (`valCompany`, etc.) for computing the valuation of the various data types.

Now consider the following type-directed function, `valuation`, that computes the total valuation of a company, in terms of the salaries it pays out:

```
valuation :: Company -> Float
valuation = everything (+) (mkQ 0 getSalary)
```

This example introduces two new type-directed operators described Lämmel and Peyton Jones (2003), `everything` and `mkQ`.[2] The function `everything` is a form of type-directed fold or query, which takes two

2. I am using these simplified operators rather than the more general type-directed operator, `gfoldl`, because according to Lämmel and Peyton Jones: "Trying to understand the type of `gfoldl` directly can lead to brain damage."

```
module Companies (
  Company, Dept, SubUnit, Employee, Person , Salary, -- exported data types
  Manager, Name, Address,                            -- exported type definitions

  -- exported accessors
  companyDepts, ..., getSalary

  -- exported constructors
  newCompany, ..., newSalary

  -- exported valuation functions
  valCompany, ..., valSalary
) where
    ...
    ...
    companyDepts :: Company -> [Dept]
    companyDepts (C ds) = ds
    ...
    getSalary :: Salary -> Float
    getSalary (S s) = s

    newCompany :: [Dept] -> Company
    newCompany ds = C ds
    ...
    newSalary :: Float -> Salary
    newSalary s = S s

    valCompany :: Company -> Float
    valCompany (C ds) = foldl (+) 0 (map valDept ds)
    ...
    valSalary :: Salary -> Float
    valSalary (S s) = s
```

Figure 1·3: A module interface for the `Company` and related data types.

functions. Its second functional argument is a query function, with type `forall a.a -> b`, for some type `b`, that is applied to all subterms of an input value (including the value itself). The first functional argument of `everything` is a combining function, with type `b -> b -> b`, that can be used to combine answers returned from the query function.

In the function `valuation`, the query function argument used by `everything` is constructed using the type-directed function `mkQ` ("make query"). The function `mkQ` takes a default value of type `b` and a monomorphic query function for a specific type, that is, a function from `c -> b`, for some type `c`, and creates a new type-directed function of type `forall a.a -> b`. When applied to values of type `c`, the function created by `mkQ` returns the value that would be computed by the monomorphic query function. For inputs of any other type, the function created by `mkQ` will return the default value. So, for example, the function created by `(mkQ 0 getSalary)` returns `2000` if applied to `(newSalary 2000)` and returns `0` if applied to `(newCompany [])` – if `(mkQ 0 getSalary)` is applied to a salary it returns the `Float` representation of that salary, otherwise it just returns the `Float` value `0`.

The function `valuation` walks over every subterm of its input using the `everything` operator, applying the function `(mkQ 0 getSalary)` to every subterm. The `everything` operator then uses the function `(+)` to combine the result of each of these applications. The overall result is that `valuation` will return the sum of all the salaries in a value of type `Company`.

It is also worthwhile to revisit the type-directed function `increase` described in the previous section. Because the data constructor `S` for `Salary` is no longer visible, the function `increase` must be rewritten as:

```
increase :: Float -> Company -> Company
increase p = everywhere (mkT (increaseS p))

increaseS :: Float -> Salary -> Salary
increaseS p s = newSalary ((getSalary s) * (1 + p))
```

Because both `valuation` and `increase` are type-directed functions, their behavior necessarily depends upon the implementation of the `Company` type in the `Companies` module. Therefore, if the implementation of `Companies` module is changed the behavior of these two functions can change. Therefore, I say that the `Companies` module does not have confidentiality with respect to the functions `valuation` and `increase`.

The problem is that because changing the implementation of the `Companies` module can alter the behavior of the `valuation` and `increase`, it is possible to naïvely make changes that can result in incorrect behavior. For example, a programmer maintaining the `Companies` module might decide that for especially large companies, it is important to cache the total payroll for a department within the data structure itself. This could be done by redefining the `Dept` data type inside the module as:

```
data Dept     = D Name Manager Float [SubUnit]
```

In this revised definition, the `Float` argument is used to cache the value of the payroll. Because this change can be made without altering the interface as defined in Figure 1·3 the programmer might assume that it is safe to make this change. However, using the type-directed function `increase` will now

corrupt values of type `Company` because it does not update cached valuations. The type-directed function `valuation`, unlike `increase`, continues to behave the same as it did before.

One might argue that, `increase`, or a function like it, should be provided as part of the abstraction provided by the `Companies` module. In practice, the author of an ADT cannot predict all the operations that could be desirable. Therefore, it can be necessary to write a function like `increase` after the fact, and without access to the ADT's source code.

It is also important to note that adding a cache to the definition of `Dept` in the previous section would have also caused values of type `Company` to become corrupted by the original version of the `increase` function. However, in the previous section the data type definitions were not abstract, so it is not reasonable to assume that changing the implementation will not impact the behavior of the program. That is, confidentiality is only defined for abstract type definitions, not concrete type definitions.

Furthermore, why is it that the `increase` function proves problematic and the `valuation` function is not, when both functions violate the confidentiality of the `Companies` module? The reason is, that in practice, representation independence and confidentiality are stronger properties than are always necessary – sometimes it will be acceptable for these properties not to hold. This proved to be the case, for example, with the `valuation` function that continued to work correctly despite the change in the `Companies` module. In the remainder of this section I will introduce a weaker property than confidentiality, called *integrity*,[3] that is violated by the function `increase`. I claim that while it is sometimes useful to violate confidentiality, integrity should always hold.

For some ADTs, there can be invariants on values of the abstract type. The problem in the example above arose because when the definition of the `Dept` type was changed, this introduced an invariant on the `Companies` module. This invariant was that the third argument to the `D` constructor is equal to the sum of the valuations of its `Manager` and `SubUnit` components. In a language without TDP, operations defined for such an ADT can safely assume that these invariants always hold on their inputs.[4] In the case of the `Companies` module, this would mean that the `valDept` function can assume that it will always receive values of type `Dept` with a correctly cached valuation. This is a reasonable assumption, because no part of the program outside the `Companies` module should be able to manipulate these values because their definitions are hidden.

By using TDP it is possible to construct values of the ADT that violate the invariants. In the example above, the type-directed function `increase` did just that. If the function `valCompany` should receive one of these values as an input, I say that the *integrity* of the `Companies` module has been violated. This also explains why the `valuation` function remains benign, despite the change to the implementation of the `Companies` module – it never produces values of an abstract type, so `valuation` can never violate any hidden invariants.

Note that I distinguish between the creation of a value that does not satisfy the invariants of the ADT, and the use of such a value by a function that expects those invariants to hold. The reason for this distinction is because it may not always be possible to atomically modify a value so that the invariant is

3. In the context of information-flow systems, the property of integrity is often considered to be a dual to the property of confidentiality (Biba 1977). I am not using the term integrity in this sense.

4. This assumption requires that the operations defined as part of the ADT itself always produce output values that meet this invariant.

guaranteed to hold during execution.[5] For example, just using the `increase` function does not violate integrity, but calling the function `valCompany` on a value produced by `increase` does violate integrity.

I claim that the property of integrity is subsumed by confidentiality, and in turn by representation independence. This is because it is only possible to violate the integrity of an ADT by having knowledge of that ADT's implementation.[6] Knowing the implementation of an ADT can only be accomplished by first violating confidentiality of that ADT. For example, the `increase` function only works because it can traverse the structure of a `Company` and its components. It can only traverse over the children of a `Dept` node because the function `everywhere` will violate the confidentiality of `Dept` to learn the structure of its implementation. Therefore, to reconcile TDP with ADTs it seems that it is important to not only be able to reason about the confidentiality of an ADT, but its integrity as well.

I arrived at the properties of confidentiality and integrity independently, but they were considered quite early in the study of data abstraction. For example, Morris in his paper *Types are not sets* (1973) says:

> All values used to represent the abstract objects are considered to be of a certain type. The rules are:
>
> - Only values of that type can be submitted for processing (authentication).
> - Only the procedures given can be applied to objects of that type (secrecy).
>
> The remaining question is how to decide whether a given value has a particular type.

Morris's notions of authentication and secrecy are analogous to my notion of integrity and confidentiality, respectively.

One solution for ensuring that integrity always hold is to simply prohibit TDP. However, TDP is an invaluable programming technique, so prohibiting TDP altogether is counter-productive. A second solution is to have all ADT operations validate their inputs to ensure that all expected invariants hold. This will ensure that integrity always holds, but can be too computationally expensive in practice. Another solution to ensure integrity, would be to always express the invariants of an ADT as part of its definition. In other words, as part of the type itself. This, however, requires a far more powerful type system than is currently outside of experimental programming languages. A third possible solution for preserving integrity is to examine whether a technique developed for securing and protecting program data can be applied to securing and protecting type meta-data as well. In the next section, I will explain how I propose to use techniques from information-flow type systems to statically reason about the confidentiality and integrity of ADTs.

5. An analogous situation arises in languages that use a substructural type systems to ensure the safety of dynamically managed memory. Languages like Cyclone (Swamy et al. 2006) and Vault (DeLine and Fähndrich 2001) allow the linearity of a memory reference to be violated within a restricted scope, as long as linearity is restored before the scope ends.

6. Confidentiality only subsumes integrity in type safe languages. In a language where unsafe and unchecked casts are allowed, it is possible to violate integrity without having knowledge of an ADT's implementation. For example, the following C++ (Stroustrup 2000) code fragment allows code like `string* foo = (string *)(new int); cout << *foo;` which violate the integrity of the `string` ADT without knowledge of its implementation.

§ 1·3 Reasoning about confidentiality and integrity using information-flow

My thesis in this dissertation is that information-flow kind and types systems can be used to reason statically about the confidentiality and integrity of ADTs in the presence of TDP. The reason for this is that information-flow kind and type systems make the dependencies in programs explicit. Because the dependencies are explicit and recorded in the types and kinds of a program, programmers can reason about how different parts of the program are related without needing to inspect all of the code itself. Because types, and the associated information-flow annotations, are known at compile time, programmers can also reason statically about their ADTs and their use of TDP in programs. A programmer could know by examining a type signature that changing her implementation of an ADT for sets could potentially affect the program's behavior without needing to make the change, run the program, and observe the behavior. Finally, information-flow kind and type systems can allow programmers not only to observe the dependencies in their program, but allow them to enforce policies on the allowable dependencies using type annotations.

The reason I chose to use an information-flow kind and type system to reason about confidentiality and integrity is because information-flow type systems have been successfully used to reason about the confidentiality and integrity of term data. Volpano, Smith, and Irvine (1996) showed that a static information-flow analysis could be formulated as type system and proved its soundness with respect to the property known as *noninterference*. In an information-flow type system the types of data are *labeled* with an information content. Usually the information content is expressed in terms of a lattice (Bell and La Padula 1975; Denning 1976). The bottom element of the lattice, $\bot$, is informally considered to be "low security" data while the top of the lattice, $\top$, is informally considered to be "high security" data.

A program is noninterfering if changing high security input values, that is, values whose types are labeled with $\top$, will not change the resulting low security output values, that is, values whose types are labeled with $\bot$. It is worth noting that this sounds very familiar to representation independence, where changing the implementation of an abstract data type does not affect the behavior of the program. In fact, a program that is noninterfering is said to preserve confidentiality of data; it is not a coincidence that I have been using the term confidentiality. It is natural to consider: If an information-flow type system can prevent high security data from affecting low security data, can an information-flow kind system prevent high security, or abstract, type meta-data from affecting the low security term and type data?

To illustrate how information-flow kind and type systems can be used observe the dependencies and enforce policies in programs with ADTs and TDP, I will start with the example program in Figure 1·4. It defines a module containing two type-directed operations, a module `Nat` implementing an ADT for natural numbers, and a module `Set` implementing an ADT for sets. The example is written in Standard ML (SML) (Milner, Tofte, Harper, and MacQueen 1997) extended with a single new type-directed operation called `typecase`. I will first explain how the `typecase` primitive works and then I will explain Figure 1·4 in detail. Finally, I will then show how the example could be extended with an information-flow kind and type system.

The `typecase` operator works very much like the SML `case` operator for pattern matching on algebraic data types, but instead of pattern matching on values, `typecase` pattern matches on types. For clarity,

```
structure Generic = struct
   fun cast (x : 'a) : 'b =
       typecase 'a of 'b => x | _  => abort "Types are not the same"

   fun eq (x : 'a) (y : 'a) : bool =
       typecase 'a of int     => x = y
                    | bool    => if then y else (not y)
                    | 'b * 'c => (eq (#1 x) (#1 y)) andalso (eq (#2 x) (#2 y))
                    | _       => abort "Cannot compare this type for equality"
end :> sig
   val cast: 'a -> 'b
   val eq:   'a -> 'a -> bool
end

structure Nat = struct
   type t = int
   val z = 0
   fun s n = n + 1
   fun pred n = if n = 0 then 0 else (n - 1)
end :> sig
   type t
   val z:     t
   val s:     t -> t
   val pred:  t -> t
end

structure Set = struct
   type 'a t = 'a list
   val empty = []
   fun member x [] = false
    |  member x (x'::xs) = (Generic.eq x x') orelse (member x xs)
   fun add x s = if (member x s) then s else (x::s)
   fun remove x [] = []
    |  remove x (x'::xs) = if (Generic.eq x x') then xs else (x'::(remove x xs))
end :> sig
   type 'a t
   val empty:  'a t
   val member: 'a -> 'a t -> bool
   val add:    'a -> 'a t -> 'a t
   val remove: 'a -> 'a t -> 'a t
end
```

Figure 1·4: An example of type-directed programming in SML using `typecase`.

I will often refer to the type that `typecase` dispatches on as the *scrutinee*. One additional difference between `case` and `typecase` is that the latter performs *type refinement*. The following example concisely illustrates type refinement:

```
fun negate (x: 'a) : 'a =
    typecase 'a
          of int  => ~x
           | bool => not x
           | _    => abort "This type cannot be negated"
```

In this example, `typecase` does not just alter the control-flow of the function. Inside each of the branches of `typecase` it also *refines* the type that it matched against. It does this by introducing a new type equality into the environment. For example, when typechecking the branch for when the type variable `'a` matches against the type `int`, the type equality `'a = int` will be assumed. Otherwise, the expression `~x` would not be well-typed – applying the function `~`, which has a type of `int -> int`, to the value `x`, with type `'a`, is not allowed. However, it is allowed because the typechecker knows that `'a` is equal to `int` inside this branch. Similarly, inside the second branch the typechecker assumes the type equality `'a = bool`.

In Figure 1·4 the Module `Generic` defines a library of two type-directed functions: `cast` and `eq`. The function `cast` is a type-safe cast that compares the type of its argument with the desired result type and, if they match, it returns the input unchanged, otherwise it aborts with an error message. The function `eq` implements a very basic version of polymorphic structural equality. For primitive types, like `int`, it calls the primitive equality function, and for compound types, like tuples `'b * 'c`, it calls itself recursively on the components. For types that it does not know how to compare, `eq` aborts.

The module `Nat` defines a minimal implementation of the natural numbers as the abstract type `Nat.t`. The value `Nat.z` is zero, `Nat.s` is the successor function, and `Nat.pred` is a predecessor function that returns zero as the predecessor of zero. The module `Nat` is ascribed with an opaque signature that does not reveal that `Nat.t` is defined in terms of integers. The module `Nat` has the unspecified invariant that `Nat.pred` will never receive a negative integer as an argument.

The module `Set` defines a simple implementation of sets. Sets themselves are represented as lists of elements, where the empty set, `Set.empty` is implemented as the empty list. The module defines set membership with the `Set.member` function; this function is especially interesting because it uses the type-directed function `Generic.eq` to test whether elements of the abstract type `'a` are equal. The module also defines functions for adding elements to a set (`Set.add`) and removing elements (`Set.remove`). Before consing the element to be to be added to the set onto its list argument, the function `Set.add` uses the function `Set.member` to determine whether the element is already in the set. Like `Set.member`, the function `Set.remove` uses the type-directed function `Generic.eq` to test whether the element, of type `'a`, at the head of the list is the same as the element to be removed. The module `Set` has the unspecified invariant that `Set.remove` will never receive a list with duplicate elements as an input.

The example in Figure 1·4 contains several confidentiality violations. The functions `Generic.cast`, `Generic.eq`, `Set.member`, `Set.add`, and `Set.remove` all violate the confidentiality of their type parameters. That is, their behavior will depend on the type parameter they are instantiated with. For example, the following program fragment will cause the execution of the program to abort:

```
Set.add (Set.add 1 Set.empty) (Set.add (Set.add 1 Set.empty) Set.empty)
```

while the following program fragment will return with a value:

```
Set.add (Nat.s Nat.z) (Set.add (Nat.s Nat.z) Set.empty)
```

The reason for this difference is that the adding elements to a set requires using `Generic.eq` to determine whether an element is already in the set. However, because `Generic.eq` does not have a case for values of type `int list`, it will abort in the first fragment. `Generic.eq` does have a case for values of type `int`, and so the second fragment executes as expected. This program fragment is not an example of a violation of the integrity of the `Set` module because `Set.remove` never receives a list containing duplicate elements.

The example code in Figure 1·4 does not contain any integrity violations, but it is straightforward to construct small examples that do violate integrity using the three modules. For example, it is possible to violate the integrity of the `Nat` module by writing the expression

```
Nat.pred (Generic.cast ~1 : Nat.t)
```

Similarly, the integrity of the `Set` module may be violated with the expression

```
Set.remove 1 (Generic.cast [1, 1] : int Set.t)
```

Both of these violations are created by using the function `Generic.cast` to bypass the type abstraction of the `Nat` and `Set` modules.

These modules are difficult for a programmer to reason about because the type signatures do not provide any information about the relationships and dependencies between them. For example, the signature for the `Set` module does not provide any indication that the operations on sets will depend upon the type of the elements. Furthermore, a programmer cannot tell whether using a different implementation of modules would cause the program to behave any differently.

Figure 1·5 shows how the example in Figure 1·4 could be written in a language with an information-flow type and kind system. Note that this example is not written in InforML, the language I will introduce § 3. It is instead a simplified realization of the same ideas, so that it is a more gradual departure from the original example in Figure 1·4.

There are five significant changes, in Figure 1·5, from the original example: explicitly specified kinds with labels, explicitly specified quantification, label polymorphism, constraints on labels, types with labels, and label creation.

Because kinds are labeled with the information content of type meta-data, unlike in Figure 1·4, the kinds of type variables can no longer be left completely implicit. I write `* @ l` for the kind of a type whose associated meta-data has an information content of `l`. It can be read as "a type *at* level `l`". I write `l1 ⊔ l2` for the *join* of the two labels `l1` and `l2`, that is, the smallest label representing an information content greater than both `l1` and `l2`.

Now that the kinds of type variables must be made explicit, it is no longer possible to always leave the universal quantification over type variables implicit, as it is in ML-like languages. I write `⟨'a :* @ l⟩` to indicate that the type variable `'a`, with the kind `* @ l`, is universally quantified in the type that follows. Furthermore, just as many ML functions are polymorphic in the types of their arguments, in a language with information-flow, functions are frequently polymorphic in their labels. Therefore, in the example, it is possible to specify that a label is universally quantified in a type by writing `⟨l⟩`. If a function is both label

```
structure Generic = struct
   newlabel eql
   fun cast ⟨ll l2 l3|('a : * @ l1) ('b : * @ l2)⟩ (x : 'a @ l3) : 'b @ (l1 ⊔ l2 ⊔ l3) =
       typecase 'a of 'b => x | _  => abort "Types are not the same"

   fun eq ⟨l1 l2|'a : * @ l1|l1 <: eql⟩ (x : 'a @ l2) (y : 'a @ l2) : bool @ (l1 ⊔ l2) =
       typecase 'a of int     => x = y
                    | bool    => if x then y else (not y)
                    | 'b * 'c => (eq (#1 x) (#1 y)) andalso (eq (#2 x) (#2 y))
                    | _       => abort "Cannot compare this type for equality"
end :> sig
   label eql
   val cast : ⟨l1 l2 l3|('a : * @ l1) ('b : * @ l2)⟩ 'a @ l3 -> 'b @ (l1 ⊔ l2 ⊔ l3)
   val eq   : ⟨l1 l2 l3|'a : * @ l1|l1 <: eql⟩ 'a @ l2 -> 'a @ l3 -> bool @ (l1 ⊔ l2 ⊔ l3)
end

structure Nat = struct
   type t = int
   val z = 0
   fun s n = n + 1
   fun pred n = if n = 0 then 0 else (n - 1)
end :> sig
   type t : * @ Generic.eql
   val z : t @ ⊥
   val s : ⟨l⟩ t @ l -> t @ l
   val pred : t @ ⊥ -> t @ ⊥
end

structure Set = struct
   type 'a t = 'a list
   val empty = []
   fun member x [] = false
    |  member x (x'::xs) = (Generic.eq x x') orelse (member x xs)
   fun add x s = if (member x s) then s else (x::s)
   fun remove x [] = []
    |  remove x (x'::xs) = if (Generic.eq x x') then xs else (x'::(remove x xs))
end :> sig
   type ('a : * @ Generic.eql) t : * @ ⊤
   val empty : ⟨l|'a : * @ l|l <: Generic.eql⟩ ('a t) @ ⊥
   val member : ⟨l|'a : * @ l|l <: Generic.eql⟩ ('a @ l2) -> (('a @ l) t) @ l -> bool @ l
   val add : ⟨l|'a : * @ l|l <: Generic.eql⟩ ('a @ l) -> ('a t) @ l -> ('a t) @ l
   val remove : ⟨l|'a : * @ l|l <: Generic.eql⟩ ('a @ l) -> ('a t) @ l -> ('a t) @ l
end
```

Figure 1·5: An example of type-directed programming augmented with information-flow.

and type polymorphic, this is specified by writing `⟨l|'a :* @ l⟩`, where there is a vertical bar between the quantified label variables and the quantified type variables.

In order for some functions to typecheck, or so to allow the programmer to specify a policy on information-flows, some polymorphic values in the example also include a constraint that serves as a precondition. For example, several functions in Figure 1·5 have a prefix like `⟨l1|'a :* @ l1|l1 <: l2⟩`. This means that it is a value that is polymorphic in the labels `l1` and `l2` and the type variable `'a`, with kind `* @ l1`, and that it has the precondition that the label `l1` must be less than or equal to the label `l2`.

Just as kinds are labeled to specify the information content of type meta-data, it is also necessary in Figure 1·5 to label types to specify the information content of term data. Again, this is done with the "at" (@) symbol. For example, in Figure 1·4, the function `Generic.cast` had type `'a -> 'b` it now has type

```
⟨l1 l2 l3|('a : * @ l1) ('b : * @ l2)⟩ 'a @ l3 -> 'b @ (l1 ⊔ l2 ⊔ l3).
```

This type can be read as "for all labels `l1` and `l2`, and for all types `'a` and `'b`, with the kinds `* @ l1` and `* @ l2` respectively, given a value of type `'a` with an information content of `l3` it will return a value of type `'b` with an information content of `l1 ⊔ l2 ⊔ l3`". However, it is worth understanding why this is the correct type for `Generic.cast`.

In the body of the `Generic.cast` function, the decision of whether `'a` should match with `'b` or match with _ depends upon the type meta-data associated with `'a` and `'b`. However, because the structure of the type `'a` has an information content of `l1` and the structure of the type `'b` has an information content of `l2`, values computed as a result of this decision must have at least an information content of `l1` and `l2`. Furthermore, the value `x` has an information content of `l3`, so the value of type `'b` that is returned by the function must also have an information content of at least `l3`. Therefore, the best information-flow annotation that can be given to result type of `Generic.cast` is `(l1 ⊔ l2 ⊔ l3)`, which means that the result necessarily depends upon the information content of its type arguments as well as the information content of the value to be cast.

Similarly, because the result of `Generic.eq` depends upon the structure of the quantified type `'a`, having an information content of `l1`, and its two arguments `x` and `y`, having an information contents of `l2` and `l3` respectively, the boolean returned must have an information content of at least `(l1 ⊔ l2 ⊔ l3)`. However, unlike `Generic.cast`, `Generic.eq` has a constraint that the quantified label `l1` must be less than or equal to the label `eql`.

The label `eql` in Figure 1·5 is a new label constant defined within the `Generic` module using the `newlabel` primitive. Therefore, in other parts of the example it is referenced using the fully qualified name `Generic.eql`. The label definition for `eql` does not specify anything about its properties, so it can only be assumed that `eql` is greater than or equal to the ⊥ label and less than or equal to the ⊤ label. The purpose of the label `eql` is to specify an upper bound on the information content of types that may be used `Generic.eq`.

The constraint `l1 <: eql` on `Generic.eq` is not required by its implementation, but is instead an example of using kind and type annotations to specify the allowable dependencies in a program. In this case, the goal of defining the label `eql` and giving `Generic.eq` the constraint annotation `l1 <: eql` is to specify that this implementation of type-directed equality may only be used to compare values of type

'a where the information content of 'a is less than or equal to eql. I will explain the impact of this policy as I describe more of the differences between Figure 1·4 and Figure 1·5.

When moving to an information-flow kind and type system, the Nat module does not require changes to its implementation. However, there several changes to its signature. First, the abstract type Nat.t has been given the kind annotation * @ Generic.eql to indicate that the type Nat.t has an information content of Generic.eql. Second, the value Nat.z has been annotated with the label ⊥ to indicate that it has no information content. Third, the function Nat.s has been made label polymorphic. Fourth, the function Nat.pred has been labeled such that it accepts only natural numbers with an information content of ⊥ and returns natural numbers with an information content ⊥.

The reason I chose these particular label annotations was to prevent the integrity violation for the Nat module I described earlier. With the information-flow kind and type system, the expression

```
Nat.pred (Generic.cast ~1 : Nat.t)
```

I gave earlier would need to be rewritten as

```
Nat.pred (Generic.cast ~1 : Nat.t @ Generic.eql)
```

Aside from the fact that types must now be labeled with an information content, it is necessary to specify that the result of using Generic.cast has an information content of Generic.eql.

As I described above, the result of Generic.cast depends upon its type arguments as well as the value being cast. If I instantiate Generic.cast with the types int @ ⊥ and Nat.t @ ⊥, and give it the argument ~1, it must necessarily return a value with an information content of Generic.eql. This is because the information content of int @ ⊥ is ⊥, the information content of Nat.t @ ⊥ is Generic.eql, and the information content of ~1 is ⊥. Therefore, the label l1 ⊔ l2 ⊔ l3 on the Generic.cast's range will be ⊥ ⊔ Generic.eql ⊔ ⊥ after instantiation, which simplifies to just Generic.eql.

If the expression Generic.cast ~1 must now have the type Nat.t @ Generic.eql, then the expression

```
Nat.pred (Generic.cast ~1 : Nat.t @ Generic.eql)
```

will no longer be well-typed. This is because Nat.pred has been annotated to accept only inputs with an information content of ⊥. Nothing is known about the label Generic.eql, so it cannot be assumed to be equivalent to ⊥.

Like the Nat module, no changes are required to the implementation of the Set module in Figure 1·5. However, there are several changes to Set's signature. First, Set.t's type parameter, 'a, is annotated to specify that it has kind Generic.eql. The reason this annotation was used is to specify that sets may only be constructed from elements with types that may be compared for equality using Generic.eq. The type Set.t itself is annotated with the kind * @ ⊤ – indicating that it has the maximal information content. Finally, all of term members of the module Set have given explicit label and type quantifiers along with the constraint that the quantified label is less than or equal to Generic.eql.

Just one benefit of the way I have chosen to annotate the kinds and types of the Set module is that it is possible to prevent the confidentiality violation I described earlier, where the behavior of the Set module could inadvertently depend on the abstract type used to implement Set.t. In the original example, the

following program fragment would cause execution to abort because `Generic.eq` does not know how to compare lists.

```
Set.add (Set.add 1 Set.empty) (Set.add (Set.add 1 Set.empty) Set.empty)
```

With the information-flow kind annotations on `Set`'s signature, `'a Set.t` now has a kind with an information content of $\top$ and the type system will statically reject the above code because sets may only contain elements whose types have an information content of `Generic.eql` or less. The label $\top$ would only be less than or equal to `Generic.eql` if `Generic.eql` were equal to $\top$, but there is not enough information available for the typechecker to determine whether that is the case. Therefore, the behavior of `Set.add` can no longer depend upon the implementation of `Set.t`.

For very similar reasons, my earlier example violating the integrity of `Set` module will be rejected during compilation by the typechecker:

```
Set.remove 1 (Generic.cast [1, 1] : ((int @ ⊥) Set.t) @ ⊤)
```

Because the result produced by `Generic.cast` necessarily depends upon the information-content of the abstract type `'a Set.t`, it must necessarily produce a value of type `(int @ ⊥) Set.t` with an information content of $\top$. The function `Set.remove` has the constraint that it will only accept inputs with an information content less than or equal to the label `Generic.eql`, so this function application is now ill-typed.

However, it is important to note that while I have managed to prevent several confidentiality and integrity violations by using an information-flow kind and type system, the annotations I have chosen still allow for valuable uses of TDP. For example, the following program fragment I gave earlier is still well-typed:

```
Set.add (Nat.s Nat.z) (Set.add (Nat.s Nat.z) Set.empty)
```

The reason that this code still typechecks is because the abstract type `Nat.t` was given a kind with an information content of `Generic.eql`. Therefore, the constraint on `Set.add` that it may only be used on sets where the element type has an information content is less than or equal to `Generic.eql` is trivially satisfied.

In this section I have only given a very informal account of how information-flow kind and type systems can be used to reason about confidentiality and integrity of ADTs and how they may be used to specify confidentiality and integrity policies. In § 2, I will provide a much more detailed and formal account of the reasoning principles that can be proven and derived for a specific instance of an information-flow kind and type system. In § 3 and § 4, I will discuss a more realistic account of programming in a language with an information-flow kind and type system and how information-flow annotations may be used to specify policies on how TDP and ADTs may interact. In the next section, I will discuss some other techniques from language based security that may be applied to the problem of reconciling TDP and data abstraction.

§ 1·4 Related work

In this section, I will show how other mechanisms from language based security may be used provide confidentiality and integrity guarantees for ADTs. My study of the literature has shown that other than information-flow techniques, the primary mechanisms for protecting data fall into two categories:

- Access control. I use access control to mean any mechanism that can be used to prevent the examination of type meta-data. Broadly, access to type meta-data can either be determined at compile-time or at runtime.
- Runtime monitoring. Protection based on runtime monitoring observes the execution of a program and halts or alters the behavior if it attempts to violates a desired policy. Runtime monitoring allows for very expressive and precise policies because it is possible to use any computable function to enforce policies on the behavior of programs.

§ Access control

Access control mechanisms simply prevent `typecase` from being used to analyze a type definition. I divide access control mechanisms into those where the access control policy for abstract types is specified at compile-time and those where the access control policy is decided at runtime.

❡ Compile-time access control

One common mechanism to statically specify whether runtime type analysis can occur is *type generativity*. Languages with type generativity allow the programmer to specify that a type is *new* or distinct from all others in the program. Technically, this does not directly prevent type analysis, but effectively it does so because these new types will only ever pattern match against themselves, which never reveals their definition.

In Figure 1·6, I have modified the original example to use a new form of signature declaration, `newtype`. This extension is most similar to the module system proposed by Govereau (2005), but his goal was to study the semantics of higher-order modules rather than limit the scope of type analysis. Therefore, there is no dynamic significance to `newtype` in his work. It is also similar to Haskell's `newtype`, where `newtype` defines a generative type, along a with a pseudo-constructor, to witness the isomorphism between the types. However, a generative type in Haskell is new at its definition, whereas the `newtype` signature in Figure 1·6 makes an existing type definition generative. Similar type generativity mechanisms have been used several times in the past to protect type abstractions (Rossberg 2003; Leifer, Peskine, Sewell, and Wansbrough 2003; Vytiniotis, Washburn, and Weirich 2005).

Giving `'a Set.t` a type signature that declares it to be generative does not affect typechecking, but will alter the behavior of the program compared to Figure 1·4. Consider my example integrity violation from the previous section:

```
Set.remove 1 (Generic.cast [1, 1] : int Set.t)
```

```
structure Generic = struct
   fun cast (x : 'a) : 'b =
       typecase 'a of 'b => x | _  => abort "Types are not the same"

   fun eq (x : 'a) (y : 'a) : bool =
       typecase 'a of int     => x = y
                    | bool    => if x then y else (not y)
                    | 'b * 'c => (eq (#1 x) (#1 y)) andalso (eq (#2 x) (#2 y))
                    | _       => abort "Cannot compare this type for equality"
end :> sig
   val cast : 'a -> 'b option
   val eq   : 'a -> 'a -> bool
end

structure Nat = struct
   type t = int
   val z = 0
   fun s n = n + 1
   fun pred n = if n = 0 then 0 else (n - 1)
end :> sig
   type t
   val z:     t
   val s:     t -> t
   val pred:  t -> t
end

structure Set = struct
   type 'a t = 'a list
   val empty = []
   fun member x [] = false
    |  member x (x'::xs) = (Generic.eq x x') orelse (member x xs)
   fun add x s = if (member x s) then s else (x::s)
   fun remove x [] = []
    |  remove x (x'::xs) = if (Generic.eq x x') then xs else (x'::(remove x xs))
end :> sig
   newtype 'a t
   val empty : 'a t
   val member : 'a -> 'a t -> bool
   val add : 'a -> 'a t -> 'a t
   val remove : 'a -> 'a t -> 'a tend
end
```

Figure 1·6: An example of type-directed programming augmented with type generativity.

This code will still typecheck, but at runtime `Generic.cast` will cause the program to abort. This is because even though `int Set.t` is implemented by the type `int list`, the fact that `'a Set.t` is considered generative means that it will not be considered equivalent at runtime by the `typecase` primitive.

However, my other example, from the previous section, of violating integrity is not prevented:

```
Nat.pred (Generic.cast ~1 : Nat.t)
```

The reason for this is that I did not declare the abstract type `Nat.t` to be generative, so at runtime the type `Nat.t` will be treated as equivalent to the type `int`. This is easily solved by changing the module signature for `Nat.t` to specify that it is generative, but that introduces another problem – it is no longer possible to create sets of natural numbers:

```
Set.add (Nat.s Nat.z) (Set.add (Nat.s Nat.z) Set.empty)
```

If `Nat.t` were declared to be generative, at runtime the above code would fail because `Generic.eq` will consider the type `Nat.t` distinct from the type `int`, and therefore abort because it does not have a case to handle values of type `Nat.t`.

So while it is possible to rule out integrity violations by using type generativity, it will also rule out potentially useful applications of TDP. It is not possible to have both, as I showed was possible using an information-flow kind and type system, because type generativity can be used to enforce confidentiality policies, but cannot be used to enforce integrity policies. Unlike an information-flow kind and type system it is not possible to express *end-to-end* policies on data using access control techniques – as soon as access has been granted it is no longer possible to enforce a policy on the data. Therefore, using generative types does not allow for the useful distinction between the properties of confidentiality and integrity.

Even though type generativity does rule out useful TDP operations, I believe that it is important to be able to provide the programmer with a mechanism for type generativity. While a programmer might implement an abstract data type for representing natural numbers via an integer, there will be occasions when it will be desirable to be able to distinguish between natural numbers and integers. As another example, the type-directed function `increase` in § 1·1 can only be written because there is a distinction between `Salary` and `Float`.

An alternate mechanism for static access control for type analysis was suggested by Harper and Morrisett (1995) in their foundational work on intensional type analysis. They propose making a distinction between analyzable types and non-analyzable types at the kind level. However, this has the same problem as type generativity because it does not allow reasoning about confidentiality and integrity separately.

Finally, *confined types* are a mechanism intended to prevent "sensitive" data from escaping a given package's scope. For example, Vitek and Bokowski (1999) show how to ensure that the random number generator used by a package for encryption does not get accidentally handed out by one of the package's public interfaces. However, confined types were developed with subtyping and a nominal object type system in mind, rather than the structural type system found in SML. In a structural setting, I conjecture that confined types would serve as a form of static access control by preventing a type from being named outside its module and values of that type from escaping the module. However, without additional

study, I cannot be certain that this interpretation is the correct one for confined types within a structural setting.

❡ Runtime access control

It is plausible that requiring access control decisions for type analysis to be predetermined at compile time is simply too inflexible, and that making the decisions at runtime would make it possible to enforce confidentiality and integrity independently. One way to implement such an access control scheme would be to require a token value to perform analysis on a type.

This form of dynamic access control is exemplified by the language λ_R. The language λ_R was developed by Crary, Weirich, and Morrisett (2002) as a way to provide a type erasure semantics for typed intermediate languages that use type analysis – an operational semantics that does not need to refer to types.

So that types do not need to be passed around at runtime, λ_R instead passes around values that represent types. The type system can still refer to all types statically, but code without access to a type's representation cannot analyze the structure of the type. Therefore, code that does not have a reference to a type's representation does not have access to the type's implementation. While the primary goal of λ_R is a type erasure semantics, the authors conjectured that, because type analysis in λ_R is tied to having access to a representation for a given type, a variant of the parametricity theorem can be recovered. Recently, Vytiniotis and Weirich (2007A; 2007B) have shown how to formalize and verify this conjecture.

Figure 1·7 shows how my running example can be rewritten so that `typecase` scrutinizes type representations rather than types. Type representations are values of type `t rep` for some type `t`. There are a small set of data constructors for values of type `t rep`, corresponding to the set of primitive types: the type `int` is represented by the data constructor `intRep`, which has type `int rep`, the type `t1 * t2` is constructed by applying the representations for the types `t1` and `t2` to the data constructor `pairRep`, which has type `'a rep * 'b rep -> ('a * 'b) rep`, etc. These data constructors are instances of what are called indexed types or GADTs (Coquand 1992; Crary and Weirich 1999; Xi, Chen, and Chen 2003; Peyton Jones, Vytiniotis, Weirich, and Washburn 2006).

Just like matching on types, matching on type representations introduces type refinements. If `typecase` is used to scrutinize a representation with type `'a rep` and it matches against the value `intRep`, the type equality `'a = int` is introduced.

All functions in Figure 1·4 that made use of type analysis now require extra parameters to receive the type representations that they need to operate. Therefore, in Figure 1·7, `Generic.cast` needs arguments for the representation of the type of the input value and the representation for the type of the desired output. `Generic.eq` requires the type representation for the values to be compared.

Additionally, for type analysis to be used on ADTs they must export a representation for the abstracted type. In Figure 1·7, `Nat` exports `Nat.rep` with type `Nat.t rep` and implements it using `intRep`.

However, all of this additional machinery does not significantly increase the expressive power over compile-time access control. The code in Figure 1·7 has the exact same properties as the code in Figure 1·6.

The confidentiality and integrity of the `Set` module is preserved because it does not export a type representation. Therefore, it is not possible to analyze instances of the type `'a Set.t`, and if it is not

```
structure Generic = struct
   fun cast (inrep : 'a rep) (outrep : 'b rep) (x : 'a) : 'b =
       typecase inrep of outrep => x
                       | _      => abort "Types are not the same"

   fun eq (rep : 'a rep) (x : 'a) (y : 'a) : bool =
       typecase rep of intRep  => x = y
                     | boolRep => if x then y else (not y)
                     | pairRep (fstrep, sndrep) =>
                         (eq fstrep (#1 x) (#1 y)) andalso (eq sndrep (#2 x) (#2 y))
                     | _ => abort "Cannot compare this type for equality"
end :> sig
   val cast : 'a rep -> 'b rep -> 'a -> 'b option
   val eq   : 'a rep -> 'a -> 'a -> bool
end

structure Nat = struct
   type t = int
   val rep = intRep
   val z = 0
   fun s n = n + 1
   fun pred n = if n = 0 then 0 else (n - 1)
end :> sig
   type t
   val rep:   t rep
   val z:     t
   val s:     t -> t
   val pred:  t -> t
end

structure Set = struct
   type 'a t = 'a list
   val empty = []
   fun member xrep x [] = false
    |  member xrep x (x'::xs) = (Generic.eq xrep x x') orelse (member xrep x xs)
   fun add xrep x s = if (member xrep x s) then s else (x::s)
   fun remove xrep x [] = []
    |  remove xrep x (x'::xs) = if (Generic.eq xrep x x') then xs else (x'::(remove xrep x xs))
end :> sig
   type 'a t
   val empty : 'a t
   val member : 'a rep -> 'a -> 'a t -> bool
   val add : 'a rep -> 'a -> 'a t -> 'a t
   val remove : 'a rep -> 'a -> 'a t -> 'a t
end
```

Figure 1·7: An example of type-directed programming using type representations.

possible to analyze them it is not possible for the program to depend upon their implementation or violate its integrity.

However, in order for it to still be possible to create sets of natural numbers, it is necessary for the `Nat` module to export a type representation. It then remains straightforward to violate the integrity of the `Nat` module in a manner similar to what I have shown before:

```
Nat.pred (Generic.cast intRep Nat.rep ~1 : Nat.t)
```

Therefore, shifting the choice of access control from compile-time to runtime has not changed the fact that access control techniques cannot enforce integrity policies.

The use of type representations is similar to work by Sumii and Pierce (2003; 2007). Initially, they showed how to use encryption to obtain parametricity results in an untyped language (2003). For encrypted data to be manipulated, it must be first decrypted. This requires the encryption key. Decryption of encrypted data using the corresponding key is isomorphic to using a type-safe cast and a type representation to obtain access to the implementation of an ADT. The correspondence is not surprising because the goal of λ_R is an untyped operational semantics, and Sumii and Pierce's goal is to reason about abstraction in an untyped setting. In their later work using bisimulations (2007), Sumii and Pierce instead use what they call *dynamic sealing*, but dynamic sealing is just encryption (or access control) under another name.

§ Runtime monitoring

A very general technique for enforcing security policies is the use of *runtime monitoring* or *execution monitoring* (Schneider 2000). In runtime monitoring, a security policy is defined by writing an auxiliary program, the monitor, that observes the execution of the program. The granularity varies from system to system; in some the monitor can observe every instruction that the program is about to execute; in others the monitor can only observe certain classes of events such as the manipulation of certain kinds of resources.

Based on the execution stream seen so far and the next pending event, the monitor can choose to terminate the program or alter its behavior in some fashion, such as raising an exception. The policies supported by runtime monitoring can be very expressive, because they are only limited to be computable functions. Another benefit of runtime monitoring is that it not necessary to modify the original program. Therefore, it is possible to develop the program and the policies independently.

When implementing monitoring policies to prevent integrity violations caused by TDP it is not always possible to completely develop the policies independently of the program. This is because the policies may need to be closely tied to the implementation. For instance, in my running example it will be necessary to add a validation function to the `Nat` module:

```
structure Nat = struct
  ...
  fun validate n = (n >= 0)
end :> sig
  ...
  val valid : t -> bool
end
```

This extension to `Nat` is necessary because only the author of the `Nat` module knows what invariants must hold for values of type `Nat.t`. Consequently, either the author of the `Nat` module must write the monitoring policy herself or export a validation function so that someone else can implement the policy. I have chosen to take the latter approach.

A popular method for implementing runtime monitoring is to use *aspect-oriented programming* (Kiczales et al. 1997). In some flavors of aspect-oriented programming, it is possible to write code, called *advice*, that will execute at specific points in the control flow of the program. To illustrate how this can be used to enforce a policy relating to TDP, here is an example of advice, written in the language AspectML (Dantas, Walker, Washburn, and Weirich 2008), that will prevent the integrity violation in the original program in Figure 1·4:

```
advice before (| Nat.pred : Nat.t -> Nat.t |) (arg: Nat.t) =
       if Nat.valid arg then
         arg
       else
         abort "Attempted to call predecessor on an invalid instance of Nat.t"
```

This first line of this code can be understood as saying "before executing `Nat.pred`, on an argument `arg` of type `Nat.t`, execute the following code". The body of the advice, the code that will be run, uses `Nat.valid` to see if the argument is a valid natural number. If it is not a valid natural number the advice will abort the program, otherwise it returns the original argument unchanged.

A more declarative means of specifying a similar policy can be achieved in the language Polymer (Bauer, Ligatti, and Walker 2004). Polymer was designed to allow programmers to enforce centralized policies on untrusted Java programs. The integrity violation in Figure 1·4 could be addressed by writing the following policy and installing it. The language extensions in the code below are based upon the functional formalization of Polymer:

```
fun query (a : action) : suggestion =
    case a
      of Act (Nat.pred : Nat.t -> Nat.t, arg) =>
           ReplaceSug
             (if Nat.valid arg then
                Nat.pred arg
              else
                abort "Attempted to call predecessor on an invalid instance of Nat.t")
       | _ => IrrelevantSug

registerpolicy query
```

In the code above, values of type `action` and `suggestion` are, respectively, actions the program monitor can observe and suggestions the policy can make to adjust the behavior of the program. A policy is expressed as a function of type `action -> suggestion` that the monitor can use to query the policy about whether some program action requires a response. The policy can then be registered with the runtime monitor using the `registerpolicy` primitive.

The policy implemented by the `query` function specifies that if the action is to apply the function `Nat.pred` to some argument `arg`, it should suggest to the monitor that it replace the call with one where the argument to `Nat.pred` is validated before the function call is made.

Polymer's highly declarative approach to policies has the advantage that it is easy to write code that will compose policies in interesting ways; it can be difficult to write advice that composes in well-defined ways. However, it would be possible to implement a Polymer style monitoring system using the primitives provided in aspect-oriented languages.

Despite the very precise and expressive policies that runtime monitoring can enforce on type analysis, there are some significant drawbacks. Schneider (2000) has shown that runtime monitoring can only enforce *safety* properties. Lamport (1977) introduced the notion of *safety* and *liveness* properties: safety properties are those that state "bad things" do not happen and liveness properties state that "good things happen eventually". Runtime monitoring is limited to enforcing safety policies, because enforcing liveness policies would require the monitor to be able to accurately predict future events.

However, information-flow policies are not expressible as safety properties (McLean 1994). Despite this result, it is possible given an information-flow policy to define a runtime monitoring policy that will enforce the policy. This monitoring policy will necessarily be more conservative in tracing information flows than the desired information-flow policy. Therefore, an information-flow type and kind system can more precisely specify and enforce confidentiality and integrity policies on type meta-data.

The second problem with runtime monitoring is a consequence of its expressiveness. With highly expressive policies programmers cannot reason statically about whether their use of TDP violates a policy. For example, the first runtime monitoring policy I described above could be rewritten as:

```
advice before (| Nat.pred : Nat.t -> Nat.t |) (arg: Nat.t) =
       if Nat.valid arg orelse isFull moon then
         arg
       else
         abort "Cannot call predecessor on invalid instance of Nat.t today"
```

Reasoning about programs with respect to this policy requires not only knowledge of the program text, but the calendar year and celestial body where the code will be executed. Furthermore, because policies enforced by runtime monitoring can be implemented and compiled separately, a programmer's only option may be to run her program and observe the behavior. At best, this approach will only tell her how the policy affects that specific execution trace. To be able to effectively reason about her software, a programmer needs to know about properties that hold for all possible executions.

Given these limitations, I do not believe programmers can successfully use runtime monitoring to reason about confidentiality and integrity properties of ADTs in the presence of TDP.

§ 1·5 Contributions

As described in the preceding section § 1·3, in this dissertation I propose to allow programmers to reason about the confidentiality and integrity of ADTs in the presence of type-directed programming by using an information-flow type and kind system. Information-flow type systems have been used in the past to provide confidentiality and integrity policies for data; the earliest work on static information flow dates

back to Denning and Denning (1977). I am the first to suggest lifting information-flow to the kind level to define confidentiality and integrity policies for type meta-data (Washburn and Weirich 2005).

This document includes the following contributions on harmoniously integrating TDP and ADTs:

- A refined analysis of the problem of representation independence in the presence of TDP using the finer-grained properties of confidentiality and integrity (§ 1·2). I discuss how information-flow kind and type systems can recover the ability to reason statically about the confidentiality and integrity of ADTs as well as enforce policies on type meta-data (§ 1·3). I also explain how access control mechanisms and runtime monitoring can be applied to the problem of enforcing confidentiality and integrity policies on type meta-data, and how they compare with the use of information-flow kind and type systems.

- A proof (§ 2 and § C) that, for a polymorphically-typed core calculus with support for runtime type analysis, an information-flow type and kind system allows a generalization of Reynold's parametricity theorem (1983). The parametricity theorem has in the past been used as a basis for reasoning about representation independence. After reviewing the proof of standard parametricity and how runtime type analysis breaks the theorem, I show how the theorem can be generalized to languages that include runtime type analysis. This generalized parametricity theorem can be used to formally reason about the confidentiality and integrity of ADTs in the presence of TDP.

- The design and implementation of a language with features including an information-flow type and kind system, runtime type generativity, runtime type analysis, and a module system (§ 3). InforML shows how the theoretical foundation of generalized parametricity can be realized in a realistic language, and provides a basis for further experimentation, and its implementation provides an executable specification of the semantics. I give a detailed introduction to programming in InforML while simultaneously providing insight into the many subtleties of its design.

- A study of programming idioms and design patterns for software written in InforML, and the reasoning principles and static guarantees the different techniques provide (§ 4). I focus on what I call *harmless reflection* (§ 4·2) and the *break and recover idiom* (§ 4·4). The harmless reflection idiom ensures that TDP cannot influence the essential behavior of a program, while the break and recover idiom allows confidentiality to be broken but integrity to be preserved.

- An overview of the implementation of the InforML language, and an examination of the most significant design trade-offs that were made while developing InforML (§ 5).

It is also worth explaining what is not addressed in this dissertation:

- I do not prove whether InforML is type-safe or has the generalized parametricity property. However, I do believe and conjecture that the implementation of InforML does have these properties. Proving type safety should be a straight-forward exercise once the relationship between subkinding and subtyping is clarified. Proving generalized parametricity for InforML is a challenging research problem in itself, but this is a consequence of language features that are orthogonal to its information-flow type and kind system. Proving parametricity (or an appropriate variation) for

realistic languages is a difficult problem. I will comment on the difficulties in proving type safety and generalized parametricity further in § 6·1.

- I do not make any claims about the confidentiality and integrity of ADTs that leave the purview of the type system. For example, the type system of InforML cannot say anything about what may happen to data written to the file-system or sent over the network. This would be an interesting practical extension to my proposed research, but I believe that the existing research by Leifer et al. (2003), and Sumii and Pierce (2003; 2007) have already solved this problem.

2

Generalizing parametricity

General principles should not be based on exceptional cases.
Robert J. Sawyer (*Calculating God*, 2000)

In the last chapter, I concluded that an information-flow type and kind system is the correct basis for reasoning about the confidentiality and integrity of abstract data types in the presence of type-directed programming. In this chapter, I introduce the core-calculus $\lambda_{\text{SEC}i}$, to formalize these ideas. After introducing $\lambda_{\text{SEC}i}$, I provide an introduction to the parametricity theorem, and how it has been used to reason about data abstraction in languages without the ability to analyze types at runtime. Finally, I show that the parametricity theorem is just a special case of a more general theorem based upon information-flow techniques.

§ 2·1 The core-calculus $\lambda_{\text{SEC}i}$

$\lambda_{\text{SEC}i}$ is a core calculus combining information flow and type analysis. The design of $\lambda_{\text{SEC}i}$ is intended to be as simple as possible while still capturing the essential interactions between data abstraction and type-directed programming. It is derived from the type-analyzing language λ_i^{ML} developed by Harper and Morrisett (1995) and the information-flow security language λ_{SEC} of Zdancewic (2002). I chose to base $\lambda_{\text{SEC}i}$ on λ_i^{ML} because it provides a simple yet expressive model of run-time type analysis. The language λ_i^{ML} was developed as an intermediate language for efficiently compiling parametric polymorphism. Similarly, λ_{SEC} was developed to study information flow in the context of the simply-typed λ-calculus.

The grammar for $\lambda_{\text{SEC}i}$ appears in Figure 2·1. It is a predicative, call-by-value polymorphic λ-calculus with booleans, functions and general recursion. Fixed points are separate from functions to make nontermination aspects of proofs modular. I have chosen to make $\lambda_{\text{SEC}i}$ predicative because it is closer in design to λ_i^{ML}, and avoids the complexities introduced by higher-order type analysis. I conjecture that

kinds	$\kappa ::= \star^{\ell}$	*types*
	$\mid \kappa_1 \xrightarrow{\ell} \kappa_2$	*operators*
type constructors	$\tau ::= \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2$	*λ-calculus*
	$\mid$ **bool**	*booleans*
	$\mid \tau_1 \to \tau_2$	*functions*
	$\mid \tau_1 \times \tau_2$	*products*
	$\mid$ **Typerec** $\tau\ \tau_{\textbf{bool}}\ \tau_{\to}\ \tau_{\times}$	*analysis*
types	$\sigma ::= (\tau)\ @\ \ell$	*injection*
	$\mid \sigma_1 \xrightarrow{\ell} \sigma_2$	*functions*
	$\mid \sigma_1 \times^{\ell} \sigma_2$	*products*
	$\mid \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$	*constructor polymorphism*
terms	$e ::=$ **true** $\mid$ **false**	*booleans*
	$\mid x \mid \lambda x{:}\sigma.e \mid e_1e_2$	*λ-calculus*
	$\mid \langle e_1,e_2\rangle \mid$ **fst** $e \mid$ **snd** e	*tuples*
	$\mid \Lambda\alpha{:}\star^{\ell}.e \mid e[\tau]$	*constructor polymorphism*
	$\mid$ **fix** $x{:}\sigma.e$	*fix-point*
	$\mid$ **if** e_1 **then** e_2 **else** e_3	*conditional*
	$\mid$ **typecase**$[\gamma.\sigma]\ \tau\ e_{\textbf{bool}}\ e_{\to}\ e_{\times}$	*analysis*
values	$v ::=$ **true** $\mid$ **false** $\mid \lambda x{:}\sigma.e \mid \langle v_1,v_2\rangle \mid \Lambda\alpha{:}\star^{\ell}.e$	
term substitutions	$\gamma ::= \cdot \mid \gamma,[e/x]$	
type substitutions	$\delta ::= \cdot \mid \delta,[\tau/\alpha]$	
term variable contexts	$\Gamma ::= \cdot \mid \Gamma,x{:}\sigma$	
type variable contexts	$\Delta ::= \cdot \mid \Delta,\alpha{:}\kappa$	

Figure 2·1: The grammar of the $\lambda_{\text{SEC}i}$ language.

my results extend to languages with impredicative polymorphism. Also for simplicity, I do not allow higher-kinded polymorphism, but conjecture that my results extend to that feature as well.

In $\lambda_{\text{SEC}i}$ type constructors, τ, which can be analyzed at run-time, are separated from types, σ, which describe terms. The language of type constructors consists of the simply-typed λ-calculus, a type operator called **Typerec**, and three primitive constructors that correspond to types: **bool**, $\tau_1 \to \tau_2$, and $\tau_1 \times \tau_2$.

§ Run-time type analysis

The term form **typecase** in $\lambda_{\text{SEC}i}$ can be used to define operations that depend on run-time type information. This term takes a constructor to scrutinize, τ, as well as three branches corresponding to the primitive constructors. As in 2·1, I will frequently use the mnemonic subscripts $\cdot_{\textbf{bool}}$, $\cdot_{\to}$, and $\cdot_{\times}$ to refer to entities that handle branches for booleans, functions types, and product types respectively.

During evaluation the constructor argument must of **typecase** be reduced to determine its head form so that a branch can be chosen.

$$\frac{\tau \rightsquigarrow^* \mathbf{bool}}{\mathbf{typecase}\,[\gamma.\sigma]\,\tau\,e_{\mathbf{bool}}\,e_{\rightarrow}\,e_{\times} \rightsquigarrow e_{\mathbf{bool}}}\ \textsc{ev:tcase-bool}$$

$$\frac{\tau \rightsquigarrow^* \tau_1 \rightarrow \tau_2}{\mathbf{typecase}\,[\gamma.\sigma]\,\tau\,e_{\mathbf{bool}}\,e_{\rightarrow}\,e_{\times} \rightsquigarrow e_{\rightarrow}[\tau_1][\tau_2]}\ \textsc{ev:tcase-arr}$$

$$\frac{\tau \rightsquigarrow^* \tau_1 \times \tau_2}{\mathbf{typecase}\,[\gamma.\sigma]\,\tau\,e_{\mathbf{bool}}\,e_{\rightarrow}\,e_{\times} \rightsquigarrow e_{\times}[\tau_1][\tau_2]}\ \textsc{ev:tcase-prod}$$

The bracketed argument to **typecase**, $[\gamma.\sigma]$, is only necessary for typechecking, so it can be ignored until I cover type checking. I write $e \rightsquigarrow e'$ to mean that term e reduces in a single step to e' and $\tau \rightsquigarrow \tau'$ to mean that constructor τ makes a weak-head reduction step to τ'. I write $\rightsquigarrow^*$ for the reflexive, transitive closure of the reduction relations. The complete dynamic semantics for $\lambda_{\mathrm{SEC}i}$ terms can be found in Definitions B·4·2 and B·4·3.

$\lambda_{\mathrm{SEC}i}$ also includes a constructor, **Typerec**, for analyzing type information. Without **Typerec**, it is impossible to assign types to some useful terms that perform type analysis (Harper and Morrisett 1995). **Typerec** implements a *paramorphism* (a type of fold) over the structure of the argument constructor. When the head of the argument is one of the three primitive constructors, **Typerec** will apply the appropriate branch to the constituent types, as well as the recursive invocation of **Typerec** on them.

$$\frac{}{\mathbf{Typerec}\,(\mathbf{bool})\,\tau_{\mathbf{bool}}\,\tau_{\rightarrow}\,\tau_{\times} \rightsquigarrow \tau_{\mathbf{bool}}}\ \textsc{whr:trec-bool}$$

$$\frac{}{\begin{array}{r}\mathbf{Typerec}\,(\tau_1 \rightarrow \tau_2)\,\tau_{\mathbf{bool}}\,\tau_{\rightarrow}\,\tau_{\times} \rightsquigarrow \tau_{\rightarrow}\,\tau_1\,\tau_2\,(\mathbf{Typerec}\,\tau_1\,\tau_{\mathbf{bool}}\,\tau_{\rightarrow}\,\tau_{\times}) \\ (\mathbf{Typerec}\,\tau_2\,\tau_{\mathbf{bool}}\,\tau_{\rightarrow}\,\tau_{\times})\end{array}}\ \textsc{whr:trec-arr}$$

$$\frac{}{\begin{array}{r}\mathbf{Typerec}\,(\tau_1 \times \tau_2)\,\tau_{\mathbf{bool}}\,\tau_{\rightarrow}\,\tau_{\times} \rightsquigarrow \tau_{\times}\,\tau_1\,\tau_2\,(\mathbf{Typerec}\,\tau_1\,\tau_{\mathbf{bool}}\,\tau_{\rightarrow}\,\tau_{\times}) \\ (\mathbf{Typerec}\,\tau_2\,\tau_{\mathbf{bool}}\,\tau_{\rightarrow}\,\tau_{\times})\end{array}}\ \textsc{whr:trec-prod}$$

The complete dynamic semantics of type constructors is given in Definition B·4·1.

§ The information content of constructors

Information-flow type systems track the flow of information by annotating types with labels that specify the information content of the terms they describe. Because type constructors have computational content in $\lambda_{\mathrm{SEC}i}$ (and influence the evaluation of terms) it is also necessary to label kinds.

Labels, ℓ, are drawn from an unspecified join semi-lattice, with a least element ($\bot$), joins ($\sqcup$) for finite subsets of elements in the lattice, and a partial order ($\sqsubseteq$). The actual lattice used by the type system is

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsc{wfc:var} \qquad \frac{}{\Delta \vdash \mathbf{bool} : \star^{\bot}}\ \textsc{wfc:bool} \qquad \frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 : \star^{\ell_2}}{\Delta \vdash \tau_1 \to \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{wfc:arr}$$

$$\frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 : \star^{\ell_2}}{\Delta \vdash \tau_1 \times \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{wfc:prod} \qquad \frac{\Delta, \alpha{:}\kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda \alpha{:}\kappa_1.\tau : \kappa_1 \xrightarrow{\bot} \kappa_2}\ \textsc{wfc:abs}$$

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 : \kappa_2 \sqcup \ell}\ \textsc{wfc:app}$$

$$\frac{\begin{array}{ll} \Delta \vdash \tau : \star^{\ell} & \Delta \vdash \tau_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\mathbf{bool}} : \kappa & \Delta \vdash \tau_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \end{array} \quad \text{where } \ell' = \mathcal{L}(\kappa) \text{ and } \ell \sqsubseteq \ell'}{\Delta \vdash \mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa}\ \textsc{wfc:trec}$$

$$\frac{\Delta \vdash \tau : \kappa_1 \qquad \kappa_1 \leq \kappa_2}{\Delta \vdash \tau : \kappa_2}\ \textsc{wfc:sub}$$

Figure 2·2: Constructor well-formedness rules for $\lambda_{\text{SEC}i}$.

determined by the desired confidentiality and integrity policies of the program. Intuitively, the higher a label is in the lattice, the more restricted the information content of a constructor or term should be. For most examples in this chapter, I use a simple two point lattice ($\bot$ for low security, $\top$ for high security) that tracks the dynamic discovery of a single type definition. In practice, any lattice with the specified structure could be used. I give one example of a practical lattice with richer internal structure in § 4·4. Another example of a rich lattice structure is the Decentralized Label Model (DLM) of Myers and Liskov (2000).

The labels on kinds describe the information content of type constructors. The kind of a constructor (and therefore its information content) is described using the judgment $\Delta \vdash \tau : \kappa$, read as "constructor τ is well-formed having kind κ with respect to the type variable context Δ". Figure 2·2 shows the definition of this judgment. The operator $\mathcal{L}(\kappa)$, defined in Figure 2·3, extracts the label of a kind.

The kind system is conservative: If the label of κ is ℓ, then the information content of a constructor of kind κ is *at most* ℓ. The information level of a constructor can be raised via subsumption. Because kinds are labeled, the ordering $\sqsubseteq$ on labels induces a sub-kinding relation, $\kappa_1 \leq \kappa_2$. A kind $\star^{\ell_1}$ is a sub-kind of $\star^{\ell_2}$ if $\ell_1 \sqsubseteq \ell_2$. Sub-kinding for function kinds is standard. The relation is reflexive and transitive by definition; the complete definition of subkinding can be found in § B·3.

The label of a constructor τ, of kind $\star^{\ell}$, also describes the information gained when the constructor is analyzed. For example, the kind of a **Typerec** constructor must be labeled at least as high as the scrutinized type constructor τ, as shown in the rule below. This requirement accounts for the fact that

Kind information	$\mathcal{L}(\star^{\ell}) \triangleq \ell$	$\mathcal{L}(\kappa_1 \xrightarrow{\ell} \kappa_2) \triangleq \ell$
Kind join	$\star^{\ell_1} \sqcup \ell_2 \triangleq \star^{(\ell_1 \sqcup \ell_2)}$	$(\kappa_1 \xrightarrow{\ell_1} \kappa_2) \sqcup \ell_2 \triangleq \kappa_1 \xrightarrow{\ell_1 \sqcup \ell_2} \kappa_2$
Type information	$\mathcal{L}((\tau) \mathbin{@} \ell) \triangleq \ell$ $\mathcal{L}(\sigma_1 \times^{\ell} \sigma_2) \triangleq \ell$	$\mathcal{L}(\sigma_1 \xrightarrow{\ell} \sigma_2) \triangleq \ell$ $\mathcal{L}(\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma) \triangleq \ell_1$
Type join	$(\tau) \mathbin{@} \ell_1 \sqcup \ell_2 \triangleq (\tau) \mathbin{@} (\ell_1 \sqcup \ell_2)$ $(\sigma_1 \times^{\ell_1} \sigma_2) \sqcup \ell_2 \triangleq \sigma_1 \times^{(\ell_1 \sqcup \ell_2)} \sigma_2$	$(\sigma_1 \xrightarrow{\ell_1} \sigma_2) \sqcup \ell_2 \triangleq \sigma_1 \xrightarrow{\ell_1 \sqcup \ell_2} \sigma_2$ $(\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma) \sqcup \ell_3 \triangleq \forall^{(\ell_1 \sqcup \ell_3)}\alpha{:}\star^{\ell_2}.\sigma$

Figure 2·3: Kind and type label operators for $\lambda_{\text{SEC}i}$.

the constructor that is equivalent to reducing the **Typerec** constructor will depend on the structure of τ.

$$\frac{\Delta \vdash \tau : \star^{\ell} \quad \Delta \vdash \tau_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \qquad \text{where } \ell' = \mathcal{L}(\kappa) \text{ and } \ell \sqsubseteq \ell'}{\Delta \vdash \tau_{\mathbf{bool}} : \kappa \quad \Delta \vdash \tau_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa} \text{WFC:TREC}$$
$$\Delta \vdash \mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa$$

By default the label on the **bool** constructor if $\bot$, as defined by WFC:BOOL in Figure 2·2. The label of the kind for function and product constructors must be at least as high as the join of its two constituent constructors. This is because the label must reflect the information content of the entire constructor.

To propagate information flows through type applications, the kinds of type functions, $\kappa_1 \xrightarrow{\ell} \kappa_2$, have a label ℓ that represents the information propagated by invoking the function. The information, ℓ, is propagated into the result of application as $\kappa_2 \sqcup \ell$. This is shorthand for relabeling κ_2 with $\mathcal{L}(\kappa_2) \sqcup \ell$. The precise definition for lifting label joins to kinds is given in Figure 2·3.

§ Tracking information flow in terms

The labels on types describe the information content of terms. I use the judgment $\Delta^{\star}; \Gamma \vdash e : \sigma$ to mean that "term e is well-formed with type σ with respect to the term context Γ and the type context $\Delta^{\star}$." Figure 2·4 shows definition of this judgment. I use the notation $\Delta^{\star}$ to denote type variable contexts restricted to variables of base kind $\star^{\ell}$ for any label ℓ. As I did for kinds, I define (in Figure 2·3) the operator $\mathcal{L}(\sigma)$ to extract the label of a type. Also, the judgment $\Delta^{\star} \vdash \sigma$ is used to indicate that "type σ is well-formed with respect to type context $\Delta^{\star}$."

Like constructors, the information content specified by labels for terms is conservative. The lattice ordering induces a subtyping judgment $\Delta^{\star} \vdash \sigma_1 \leq \sigma_2$, and subsumption can be used to raise the information level of a term; the complete definition of subtyping can be found in § B·3.

The types of $\lambda_{\text{SEC}i}$ include the standard ones for functions $\sigma_1 \xrightarrow{\ell} \sigma_2$, products $\sigma_1 \times^{\ell} \sigma_2$, and quantified types $\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$, plus those that are computed by type constructors $(\tau) \mathbin{@} \ell$. The rules for the well-formedness of types can be found in Figure 2·5. Note that in the well-formedness rule for types formed

$$\frac{\Delta^\star \vdash \Gamma}{\Delta^\star;\Gamma \vdash \mathbf{true} : (\mathbf{bool})\ @\ \bot}\ \textsc{wft:true} \qquad \frac{\Delta^\star \vdash \Gamma}{\Delta^\star;\Gamma \vdash \mathbf{false} : (\mathbf{bool})\ @\ \bot}\ \textsc{wft:false}$$

$$\frac{\Delta^\star \vdash \Gamma \qquad x : \sigma \in \Gamma}{\Delta^\star;\Gamma \vdash x : \sigma}\ \textsc{wft:var} \qquad \frac{\Delta^\star;\Gamma,x{:}\sigma_1 \vdash e : \sigma_2 \qquad \Delta^\star \vdash \sigma_1}{\Delta^\star;\Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \xrightarrow{\bot} \sigma_2}\ \textsc{wft:abs}$$

$$\frac{\Delta^\star;\Gamma \vdash e_1 : \sigma_1 \xrightarrow{\ell} \sigma_2 \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma_1}{\Delta^\star;\Gamma \vdash e_1 e_2 : \sigma_2 \sqcup \ell}\ \textsc{wft:app} \qquad \frac{\Delta^\star,\alpha{:}\star^\ell;\Gamma \vdash e : \sigma}{\Delta^\star;\Gamma \vdash \Lambda\alpha{:}\star^\ell.e : \forall^\bot \alpha{:}\star^\ell.\sigma}\ \textsc{wft:tabs}$$

$$\frac{\Delta^\star;\Gamma \vdash e : \forall^\ell \alpha{:}\star^{\ell'}.\sigma \qquad \Delta^\star \vdash \tau : \star^{\ell'}}{\Delta^\star;\Gamma \vdash e[\tau] : \sigma[\tau/\alpha] \sqcup \ell}\ \textsc{wft:tapp} \qquad \frac{\Delta^\star;\Gamma \vdash e_1 : \sigma_1 \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma_2}{\Delta^\star;\Gamma \vdash \langle e_1,e_2\rangle : \sigma_1 \times^\bot \sigma_2}\ \textsc{wft:pair}$$

$$\frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \times^\ell \sigma_2}{\Delta^\star;\Gamma \vdash \mathbf{fst}\, e : \sigma_1 \sqcup \ell}\ \textsc{wft:fst} \qquad \frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \times^\ell \sigma_2}{\Delta^\star;\Gamma \vdash \mathbf{snd}\, e : \sigma_2 \sqcup \ell}\ \textsc{wft:snd} \qquad \frac{\Delta^\star;\Gamma,x{:}\sigma \vdash e : \sigma \qquad \Delta^\star \vdash \sigma}{\Delta^\star;\Gamma \vdash \mathbf{fix}\ x{:}\sigma.e : \sigma}\ \textsc{wft:fix}$$

$$\frac{\Delta^\star;\Gamma \vdash e_1 : (\mathbf{bool})\ @\ \ell \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma \qquad \Delta^\star;\Gamma \vdash e_3 : \sigma}{\Delta^\star;\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \sigma \sqcup \ell}\ \textsc{wft:if}$$

$$\frac{\begin{array}{ccc}\Delta^\star \vdash \tau : \star^\ell & & \\ \Delta^\star,\gamma{:}\star^\ell \vdash \sigma & \Delta^\star;\Gamma \vdash e_{\mathbf{bool}} : \sigma[\mathbf{bool}/\gamma] & \\ \ell \sqsubseteq \ell' & \Delta^\star;\Gamma \vdash e_\to : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \to \beta/\gamma] & \text{where } \ell' = \mathcal{L}(\sigma[\tau/\gamma]) \\ & \Delta^\star;\Gamma \vdash e_\times : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \times \beta/\gamma] & \end{array}}{\Delta^\star;\Gamma \vdash \mathbf{typecase}\,[\gamma.\sigma]\,\tau\, e_{\mathbf{bool}}\, e_\to\, e_\times : \sigma[\tau/\gamma]}\ \textsc{wft:tcase}$$

$$\frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \qquad \Delta^\star \vdash \sigma_1 \le \sigma_2}{\Delta^\star;\Gamma \vdash e : \sigma_2}\ \textsc{wft:sub}$$

Figure 2·4: Term well-formedness rules for $\lambda_{\mathrm{SEC}i}$.

$$\frac{\Delta^\star \vdash \tau : \star^{\ell_1}}{\Delta^\star \vdash (\tau)\ @\ \ell_2}\ \textsc{wftp:con} \qquad \frac{\Delta^\star \vdash \sigma_1 \qquad \Delta^\star \vdash \sigma_2}{\Delta^\star \vdash \sigma_1 \xrightarrow{\ell} \sigma_2}\ \textsc{wftp:arr} \qquad \frac{\Delta^\star \vdash \sigma_1 \qquad \Delta^\star \vdash \sigma_2}{\Delta^\star \vdash \sigma_1 \times^\ell \sigma_2}\ \textsc{wftp:prod}$$

$$\frac{\Delta^\star,\alpha{:}\star^{\ell_1} \vdash \sigma}{\Delta^\star \vdash \forall^{\ell_2}\alpha{:}\star^{\ell_1}.\sigma}\ \textsc{wftp:all}$$

Figure 2·5: Type well-formedness rules for $\lambda_{\mathrm{SEC}i}$.

from type constructors,

$$\frac{\Delta^\star \vdash \tau : \star^{\ell_1}}{\Delta^\star \vdash (\tau)\ @\ \ell_2}\ \textsc{wftp:con}$$

there is no need for a connection between the label ℓ on the kind and the label on the type. That is because ℓ describes the information content of τ, while the label ℓ' on $(\tau)\ @\ \ell'$ describes the information content of a term with type $(\tau)\ @\ \ell'$. It is sound to discard ℓ, because once a constructor has been coerced to a type it can only be used statically to describe terms and cannot be analyzed.

Information flow is tracked at the term level analogously to the type level. Term abstractions, of type $\sigma_1 \xrightarrow{\ell} \sigma_2$, like type functions, propagate some information ℓ when applied. Similarly, type abstractions, $\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$, propagate some information ℓ_1 when applied. The label ℓ_2 describes the information content of constructors that can be used to instantiate the type abstraction. For products, $\sigma_1 \times^\ell \sigma_2$, the label ℓ indicates the information propagated when one of its components is projected.[1]

Like **Typerec**, the label ℓ' on the type of the **typecase** expression must be at least as high in the lattice as the label ℓ on the scrutinee. This is to account for the information learned when **typecase** examines the structure of the scrutinee.

$$\frac{\begin{array}{ccc}\Delta^\star \vdash \tau : \star^\ell & & \Delta^\star;\Gamma \vdash e_{\mathbf{bool}} : \sigma[\mathbf{bool}/\gamma] \\ \Delta^\star,\gamma{:}\star^\ell \vdash \sigma & \Delta^\star;\Gamma \vdash e_\rightarrow : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \to \beta/\gamma] & \text{where } \ell' = \mathcal{L}(\sigma[\tau/\gamma]) \\ \ell \sqsubseteq \ell' & \Delta^\star;\Gamma \vdash e_\times : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \times \beta/\gamma] & \end{array}}{\Delta^\star;\Gamma \vdash \mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathbf{bool}}\ e_\rightarrow\ e_\times : \sigma[\tau/\gamma]}\ \textsc{wft:tcase}$$

Unlike some other formulations of type analysis, $\lambda_{\mathrm{SEC}i}$'s **typecase** primitive does not introduce type equalities. For example, while typechecking $e_{\mathbf{bool}}$ it will not be the case that $\tau = \mathbf{bool} : \star^\ell$. Instead, $\lambda_{\mathrm{SEC}i}$ relies on the fact that **typecase**'s allows the type of its branches to depend upon the type it scrutinizes. That is, $e_{\mathbf{bool}}$ can produce a value of type $\sigma[\mathbf{bool}/\gamma]$, while $e_\rightarrow$ can produce a value of type $\sigma[\alpha \to \beta/\gamma]$, and $e_\times$ can produce a value of the type $\sigma[\alpha \times \beta/\gamma]$.

Because the type of a **typecase** term can depend upon the scrutinized constructor τ, it is not possible to deterministically synthesize its type solely from its subterms, τ, $e_{\mathbf{bool}}$, $e_\rightarrow$, and $e_\times$. Therefore an annotation, $[\gamma.\sigma]$, is required for typechecking **typecase**.

§ Soundness

Definition 2·1·1 (Nontermination)**.** *If* $\cdot;\cdot \vdash e : \sigma$ *and there does not exist a derivation* $e \rightsquigarrow^* v$ *then* $e \uparrow$.

$\lambda_{\mathrm{SEC}i}$ has the basic property expected from a typed language, that well-typed programs will not go wrong (Wright and Felleisen 1994).

Theorem 2·1·2 (Type Safety)**.** *If* $\cdot;\cdot \vdash e : \sigma$ *then either* e *diverges (i.e.* $e \uparrow$*), or* e *evaluates to a value that is well-typed with the type of* e *(i.e.* $e \rightsquigarrow^* v$ *where* $\cdot;\cdot \vdash v : \sigma$*).*

1. In the case of a pure functional language with only extensional equality the labels on functions, type abstractions, and products are technically unnecessary. For functions and type abstractions the information content can always be pushed into their range, and the information content of products can always be pushed into their components. In impure languages, and languages with pointer equality on values, the labels are necessary. The labels are present in $\lambda_{\mathrm{SEC}i}$ to avoid specializing too early.

$$\frac{\alpha \mapsto R \in \eta \qquad v_1 R v_2}{\eta \vdash v_1 \sim v_2 : \alpha}\ \textsc{lr:var} \qquad \frac{}{\eta \vdash v \sim v : \mathbf{bool}}\ \textsc{lr:bool}$$

$$\frac{\forall(\eta \vdash e_1 \approx e_2 : \sigma_1).\eta \vdash v_1 e_1 \approx v_2 e_2 : \sigma_2}{\eta \vdash v_1 \sim v_2 : \sigma_1 \to \sigma_2}\ \textsc{lr:arr}$$

$$\frac{\eta \vdash \mathbf{fst}\, v_1 \approx \mathbf{fst}\, v_2 : \sigma_1 \qquad \eta \vdash \mathbf{snd}\, v_1 \approx \mathbf{snd}\, v_2 : \sigma_2}{\eta \vdash v_1 \sim v_2 : \sigma_1 \times \sigma_2}\ \textsc{lr:prod}$$

$$\frac{\forall\tau_1,\tau_2.\forall(R \in \tau_1 \leftrightarrow \tau_2).\eta,\alpha \mapsto R \vdash v_1[\tau_1] \approx v_2[\tau_2] : \sigma \qquad R \text{ consistent}}{\eta \vdash v_1 \sim v_2 : \forall\alpha{:}\star.\sigma}\ \textsc{lr:all}$$

$$\frac{e_1 \rightsquigarrow^* v_1 \qquad e_2 \rightsquigarrow^* v_2 \qquad \eta \vdash v_1 \sim v_2 : \sigma}{\eta \vdash e_1 \approx e_2 : \sigma}\ \textsc{lr:term} \qquad \frac{e_1 \uparrow \qquad e_2 \uparrow}{\eta \vdash e_1 \approx e_2 : \sigma}\ \textsc{lr:divr}$$

Figure 2·6: Logically related terms in the polymorphic λ-calculus.

Proof. The theorem is proven syntactically as a corollary of the standard progress and preservation lemmas. More details can be found in Appendix C. □

§ 2·2 Generalized parametricity

The parametricity theorem has long been used to reason about programs in languages with parametric polymorphism (Reynolds 1974, 1983). For example, the theorem can be used to show that different implementations of an abstract datatype do not influence the behavior of the program or to show that external modules cannot forge values of abstract types. These are only a few of the corollaries of the parametricity theorem. This subsection starts with an overview of the standard parametricity theorem, and then examines how it can be generalized for $\lambda_{\text{SEC}i}$.

§ Parametricity

For expository purposes, this subsection and the following subsection only consider the core of $\lambda_{\text{SEC}i}$ without type constructors, security labels, or type analysis. That is, I consider a simple predicative polymorphic λ-calculus (Girard 1972; Reynolds 1974). None of the results presented in these sections are new. Informally, given a logical relation inductively defined on types, the parametricity theorem states that well-typed expressions, after applying related substitutions for their free type and term variables, are related to themselves by the logical relation. The power of the theorem comes from the fact that terms typed by universally quantified type variables can be related by any relation.

The logical relation used by the parametricity theorem is defined in Figure 2·6. Terms are related with the judgment $\eta \vdash e_1 \approx e_2 : \sigma$, read as "terms e_1 and e_2 are related at type σ with respect to the relations in η." Terms are related if they evaluate to related values, or both diverge.

$$\frac{\forall \alpha{:}\star \in \Delta^\star.(\eta(\alpha) \in \delta_1(\alpha) \leftrightarrow \delta_2(\alpha))}{\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star}\ \textsc{tslr:base} \qquad \frac{\forall x{:}\sigma \in \Gamma.(\eta \vdash \gamma_1(x) \approx \gamma_2(x) : \sigma)}{\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma}\ \textsc{slr:base}$$

Figure 2·7: Related substitutions in the polymorphic λ-calculus.

The judgment $\eta \vdash v_1 \sim v_2 : \sigma$ means that "values v_1 and v_2 are related at type σ with respect to the relations in η". The relation between values is defined inductively over types σ, potentially containing free type variables. To account for these variables, the relations are parametrized by a map, η, from type variables to binary relations on values. This map is used when σ is a type variable (see rule LR:VAR). If σ is **bool**, the relation is identity. Typical for logical relations, values of function type are related only if, when applied to related arguments, they produce related results. Likewise, values of product types are related if the projections of their components are related.

The most important rule, LR:ALL, defines the relationship between values of type $\forall \alpha{:}\star.\sigma$. Polymorphic values are related if their instantiations with *any* pair of types are related. Furthermore, *any* consistent relation R between values of those types as the relation on α can be used. I use the notation $R \in \tau_1 \leftrightarrow \tau_2$ to mean that R is a binary relation between values with the closed type τ_1 and values with the closed type τ_2. The properties of a consistent relation are dependent upon the details of the language and the proof. My requirements for consistency are very easy to meet, but I will wait until it is required by the proofs to explain them. If quantification over types of higher kind were allowed, R would have to be a function on relations. This extension is orthogonal to my result, so I restrict myself to polymorphism over kind $\star$.

To state the parametricity theorem, the notion of related substitutions for types and related terms must be defined. In Figure 2·7, the rule TSLR:BASE states that a relation mapping η is well-formed with respect to two type substitutions $\delta_1$and δ_2 for the variables in the type context $\Delta^\star$. There are no restrictions on the range of the type substitutions. On the other hand, SLR:BASE requires that a pair of term substitutions for the variables in Γ must map to related terms. Even though $\lambda_{\text{SEC}i}$ has a call-by-value semantics, term substitutions must map to terms, not values. Otherwise, it would it be impossible to prove the case for fixed points, which requires a term substitution.

With these definitions it is possible to state the parametricity theorem for my restricted language:

Theorem 2·2·1 (Parametricity)**.** *If* $\Delta^\star; \Gamma \vdash e : \sigma$ *and*
$\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ *and*
$\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma$*, then*
$\eta \vdash \delta_1(\gamma_1(e)) \approx \delta_2(\gamma_2(e)) : \sigma$.

Proof. By induction on the typing judgment with appeals to supporting lemmas. □

One complication in this proof arises in the case for type application, where I would like to show that a term $v[\tau]$ is related to itself (after appropriate substitutions) at type $\sigma[\tau/\alpha]$. By the induction hypothesis, I know that v is related to itself at type $\forall \alpha{:}\star.\sigma$, so by inversion of the rule LR:ALL I can conclude that $v[\tau]$ is related to itself at type σ, where the type α is mapped to any relation R. However, what I

$$\frac{}{\mathbf{true} \preceq \mathbf{true}}\ \text{ER:TRUE} \qquad \frac{}{\mathbf{false} \preceq \mathbf{false}}\ \text{ER:FALSE} \qquad \frac{}{x \preceq x}\ \text{ER:VAR} \qquad \frac{e_1 \preceq e_2}{\lambda x{:}\sigma.e_1 \preceq \lambda x{:}\sigma.e_2}\ \text{ER:ABS}$$

$$\frac{e_1 \preceq e_3 \qquad e_2 \preceq e_4}{e_1 e_2 \preceq e_3 e_4}\ \text{ER:APP} \qquad \frac{e_1 \preceq e_3 \qquad e_2 \preceq e_4}{\langle e_1, e_2\rangle \preceq \langle e_3, e_4\rangle}\ \text{ER:PAIR} \qquad \frac{e_1 \preceq e_2}{\mathbf{fst}\, e_1 \preceq \mathbf{fst}\, e_2}\ \text{ER:FST}$$

$$\frac{e_1 \preceq e_2}{\mathbf{snd}\, e_1 \preceq \mathbf{snd}\, e_2}\ \text{ER:SND} \qquad \frac{e_1 \preceq e_2}{\Lambda\alpha{:}\star.e_1 \preceq \Lambda\alpha{:}\star.e_2}\ \text{ER:TABS} \qquad \frac{e_1 \preceq e_2}{e_1[\tau] \preceq e_2[\tau]}\ \text{ER:TAPP}$$

$$\frac{e_1 \preceq e_2}{\mathbf{fix}\ x{:}\sigma.e_1 \preceq \mathbf{fix}\ x{:}\sigma.e_2}\ \text{ER:FIX} \qquad \frac{e_1 \preceq e_2}{\mathbf{fix}_n\ x{:}\sigma.e_1 \preceq \mathbf{fix}_n\ x{:}\sigma.e_2}\ \text{ER:FIXN1}$$

$$\frac{e_1 \preceq e_2}{\mathbf{fix}_n\ x{:}\sigma.e_1 \preceq \mathbf{fix}\ x{:}\sigma.e_2}\ \text{ER:FIXN2} \qquad \frac{e_1 \preceq e_4 \qquad e_2 \preceq e_5 \qquad e_3 \preceq e_6}{\mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 \preceq \mathbf{if}\, e_4\, \mathbf{then}\, e_5\, \mathbf{else}\, e_6}\ \text{ER:IF}$$

Figure 2·8: The erasure relation.

need to show is that $v[\tau]$ is related to itself at type $\sigma[\tau/\alpha]$. The trick is to instantiate R with the relation $\{(v_1, v_2) \mid \eta \vdash v_1 \approx v_2 : \tau\}$ and use the following type substitution lemma.

Lemma 2·2·2 (Type substitution for parametricity)**.**
If $\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ *then*
$\eta \vdash e_1 \approx e_2 : \sigma[\tau/\alpha]$ *iff*
$\eta, \alpha \mapsto R \vdash e_1 \approx e_2 : \sigma$, *where*
R *is the relation* $\{(v_1, v_2) \mid \eta \vdash v_1 \approx v_2 : \tau\}$ *and* $\delta_i(\alpha) = \delta_i(\tau)$.

Proof. The proof in both directions of the biconditional is by induction on the structure of the term relation. □

Another significant complication in the proof of Theorem 2·2·1 is circularity in relating fix-points. To show that **fix** $x{:}\sigma.e$ is related to itself I must show that e is related to itself under an extended term substitution where $\gamma_1(x) = \gamma_1(\mathbf{fix}\ x{:}\sigma.e)$ and $\gamma_2(x) = \gamma_2(\mathbf{fix}\ x{:}\sigma.e)$. However, for these substitutions to be related, I need to know that the fixed point is related to itself. But showing that the fixed point is related to itself is exactly what I am trying to show! To escape this circularity I apply a syntactic technique from Pitts (2005). I define a bounded fixed point expression that can only be unfolded a finite number of times before diverging The term $\mathbf{fix}_{n+1}\ x{:}\sigma.e$ unwinds to $e[(\mathbf{fix}_n\ x{:}\sigma.e)/x]$. By definition $\mathbf{fix}_0\ x{:}\sigma.e$ always diverges.[2]

Now that fixed points may be annotated with an index, I can define a partial order on terms called the erasure relation. The definition of this relation is given in Figure 2·8. The relation orders terms by whether fixed point expressions are annotated. The granularity of the order could be made finer by also

2. It might be more aesthetically pleasing in future presentations of this proof to instead use a single bounded fixed point operator and use a bound of ω for what I write as **fix** $x{:}\sigma.e$. This more accurately characterizes the unannotated fixed point as a "limit".

ordering fixed point expressions by their bound, but it is unnecessary for my proofs. For example, the order $\mathbf{fix}_1\ y{:}\mathbf{bool}.\mathbf{true} \preceq \mathbf{fix}\ y{:}\mathbf{bool}.\mathbf{true}$ holds but $\mathbf{fix}_1\ y{:}\mathbf{bool}.\mathbf{true} \preceq \mathbf{fix}_2\ y{:}\mathbf{bool}.\mathbf{true}$ does not.

An important property of fixed point expressions is that if a fixed point expression reduces to a value, then it must have unfolded itself a finite number of times. The following lemma formalizes this property.

Lemma 2·2·3 (Unwinding evaluation equivalence)**.**

$$\mathbf{fix}\ x{:}\sigma.e' \rightsquigarrow^* v \textit{ iff exists } n \textit{ such that for all } m{,}m \geq n \textit{ implies } \mathbf{fix}_m\ x{:}\sigma.e' \rightsquigarrow^* v' \textit{ where } v' \preceq v.$$

Proof. Both directions follow by straightforward induction over the number of reduction steps. □

At this point I can define my notion of consistency: only those relations R that cannot depend upon finite approximations of fixed points can be quantified over. More precisely, if $v_1 R v_2$ and v_1' is an erasure of v_2 and v_2' is an erasure of v_2 then R must also relate v_1' and v_2'. For example, the relation

$$\{(\lambda x{:}\mathbf{bool}.\mathbf{fix}_{n_1}\ y{:}\mathbf{bool}.\mathbf{true}{,}\lambda x{:}\mathbf{bool}.\mathbf{fix}_{n_2}\ y{:}\mathbf{bool}.\mathbf{true}) \mid n_1 = n_2\},$$

is not consistent because it will relate $\lambda x{:}\mathbf{bool}.\mathbf{fix}_7\ y{:}\mathbf{bool}.\mathbf{true}$ and $\lambda x{:}\mathbf{bool}.\mathbf{fix}_7\ y{:}\mathbf{bool}.\mathbf{true}$, but not $\lambda x{:}\mathbf{bool}.\mathbf{fix}\ y{:}\mathbf{bool}.\mathbf{true}$ and $\lambda x{:}\mathbf{bool}.\mathbf{fix}_7\ y{:}\mathbf{bool}.\mathbf{true}$.

The logical relation itself is closed under erasure, making it a consistent relation.

Lemma 2·2·4 (Logical relation is closed under erasure)**.**

- *If* $\eta \vdash v_1 \sim v_2 : \tau$ *and* $v_1 \preceq v_1'$ *and* $v_2 \preceq v_2'$ *then* $\eta \vdash v_1' \sim v_2' : \tau$
- *If* $\eta \vdash e_1 \approx e_2 : \tau$ *and* $e_1 \preceq e_1'$ *and* $e_2 \preceq e_2'$ *then* $\eta \vdash e_1' \approx e_2' : \tau$

Proof. The proof follows by straightforward mutual induction over the structure of $\eta \vdash v_1 \sim v_2 : \tau$ and $\eta \vdash e_1 \approx e_2 : \tau$. □

It is now straightforward to show that, for any n, $\mathbf{fix}_n\ x{:}\sigma.e$ is related to itself. Then the following continuity lemma can be used to prove that unbounded fixed points are related to themselves.

Lemma 2·2·5 (Continuity)**.** *If* $\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ *and*
for all n, $\eta \vdash \mathbf{fix}_n\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_n\ x{:}\sigma_2.e_2 : \sigma$
where $\delta_1(\sigma) = \sigma_1$ *and* $\delta_2(\sigma) = \sigma_2$ *then*
$\eta \vdash \mathbf{fix}\ x{:}\sigma_1.e_1 \approx \mathbf{fix}\ x{:}\sigma_2.e_2 : \sigma$.

Proof. There are four cases.

- If both $\mathbf{fix}\ x{:}\sigma_i.e_i$ diverge, they are trivially related by LR:DIVR.
- If both $\mathbf{fix}\ x{:}\sigma_i.e_i$ converge to a value, they must do so with some finite number of unwindings as specified by Lemma 2·2·3, m. It is possible to instantiate the assumption, for all n, $\eta \vdash \mathbf{fix}_n\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_n\ x{:}\sigma_2.e_2 : \sigma$, accordingly, to obtain the a derivation $\eta \vdash \mathbf{fix}_m\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_m\ x{:}\sigma_2.e_2 : \sigma$. By inversion this means either both $\mathbf{fix}_m\ x{:}\sigma_i.e_i$ diverge or converge to related values, $\eta \vdash v_1 \sim v_2 : \sigma$. However, they must converge after at most $m - 1$ unwindings, therefore it is the case that they converge to related values. Furthermore, $\mathbf{fix}\ x{:}\sigma_i.e_i$ evaluates to v_i', which is an erasure of v_i.

Because the logical relation is closed under erasure, it is the case that $\eta \vdash v_1' \sim v_2' : \sigma$. Finally because both $\mathbf{fix}\ x{:}\sigma_i.e_i$ converge to v_i' the rule LR:TERM can be used to conclude $\eta \vdash \mathbf{fix}\ x{:}\sigma_1.e_1 \approx \mathbf{fix}\ x{:}\sigma_2.e_2 : \sigma$.

- In the last two cases, one of $\mathbf{fix}\ x{:}\sigma_i.e_i$ diverges and the other converges to a value. However, the fixed point that converged must do so in a finite number of unwindings m, as described by Lemma 2·2·3. Then instantiating for all n, $\eta \vdash \mathbf{fix}_n\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_n\ x{:}\sigma_2.e_2 : \sigma$ with m I have a derivation that $\eta \vdash \mathbf{fix}_m\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_m\ x{:}\sigma_2.e_2 : \sigma$. By inversion I know that either both $\mathbf{fix}_m\ x{:}\sigma_i.e_i$ converge or diverge. However, I already know that one of the expressions converges, therefore the other must as well. However, I know that $\mathbf{fix}_n\ x{:}\sigma_i.e_i$ terminates iff $\mathbf{fix}\ x{:}\sigma_i.e_i$ does. This contradicts the assumption that only one of the two fixed points converged to a value.

□

§ Applications of the parametricity theorem

The parametricity theorem has been used for many purposes, most famously for deriving *free theorems* about functions in the polymorphic λ-calculus, from their types alone (Wadler 1989). My purpose is more similar to that of Reynolds (1974; 1983): reasoning about representation independence properties.

Corollaries of Theorem 2·2·1 provide important results for reasoning about abstract types in programs. Many specific properties can be proven as a consequence of the parametricity theorem, but I believe the following two are representative of what a programmer desires.

This first corollary says that a programmer is free to change the implementation of an abstract type without affecting the behavior of a program. It is the essence behind parametric polymorphism – type information is not allowed to influence program execution, and values of abstract type are be treated as "black boxes".

Corollary 2·2·6 (Confidentiality)**.** *If* $\cdot \vdash v_1 : \tau_1$ *and*
$\cdot \vdash v_2 : \tau_2$, *then*
$\alpha{:}\star; x{:}\alpha \vdash e : \mathbf{bool}$ *and*
$e[\tau_1/\alpha][v_1/x] \rightsquigarrow^* v$ *iff* $e[\tau_2/\alpha][v_2/x] \rightsquigarrow^* v$.

Proof. First construct a derivation that $\cdot;\cdot \vdash \Lambda\alpha{:}\star.\lambda x{:}\alpha.e : \forall\alpha{:}\star.\alpha \to \mathbf{bool}$ using the appropriate typing rules and then appeal to Theorem 2·2·1 to obtain

$$\cdot \vdash \Lambda\alpha{:}\star.\lambda x{:}\alpha.e \sim \Lambda\alpha{:}\star.\lambda x{:}\alpha.e : \forall\alpha{:}\star.\alpha \to \mathbf{bool}.$$

Next, by inversion on LR:ALL and instantiation with the relation

$$R = \{(v_1, v_2) \mid (\cdot;\cdot \vdash v_1 : \tau_1), (\cdot;\cdot \vdash v_2 : \tau_2)\},$$

it can be concluded that

$$\cdot, \alpha \mapsto R \vdash (\Lambda\alpha{:}\star.\lambda x{:}\alpha.e)[\tau_1] \approx (\Lambda\alpha{:}\star.\lambda x{:}\alpha.e)[\tau_2] : \alpha \to \mathbf{bool}.$$

By straightforward application of LR:VAR it is possible to conclude

$$\cdot, \alpha \mapsto R \vdash v_1 \sim v_2 : \alpha,$$

so by application of LR:TERM, inversion on LR:ARR, and instantiation

$$\cdot,\alpha \mapsto R \vdash (\Lambda\alpha{:}\star.\lambda x{:}\alpha.e)[\tau_1]v_1 \approx (\Lambda\alpha{:}\star.\lambda x{:}\alpha.e)[\tau_2]v_2 : \textbf{bool}.$$

Finally, because the relation is closed under reduction I have LR:ARR, and by instantiation it is true that

$$\cdot,\alpha \mapsto R \vdash e[\tau_1/\alpha][v_1/x] \approx e[\tau_2/\alpha][v_2/x] : \textbf{bool},$$

from which the desired conclusion can be obtained by simple inversion. □

This second corollary states that there is no way for a program to invent values of an abstract type, and thereby allowing the integrity of the abstraction to be violated. The integrity of the abstraction can be thought of as unspecified invariants.

Corollary 2·2·7 (Integrity)**.** *If* $\alpha{:}\star;\cdot \vdash e : \alpha$ *then* $e[\tau/\alpha]$ *for any* τ *must diverge.*

Proof. First construct a derivation that $\cdot;\cdot \vdash \Lambda\alpha{:}\star.e : \forall\alpha{:}\alpha$ using the appropriate typing rules, then appeal to Theorem 2·2·1 to obtain

$$\cdot \vdash \Lambda\alpha{:}\star.e \sim \Lambda\alpha{:}\star.e : \forall\alpha{:}\star.\alpha.$$

Now assume an arbitrary τ. By inversion on LR:ALL and by instantiation it is possible to conclude

$$\cdot,\alpha \mapsto \varnothing \vdash (\Lambda\alpha{:}\star.e)[\tau] \approx (\Lambda\alpha{:}\star.e)[\tau] : \alpha.$$

Because the relation is closed under reduction it is true that

$$\cdot,\alpha \mapsto \varnothing \vdash e[\tau/\alpha] \approx e[\tau/\alpha] : \alpha.$$

Furthermore, by inversion either $e[\tau/\alpha] \rightsquigarrow^* v$ or $e[\tau/\alpha]\uparrow$. However in the former case that would mean that

$$\cdot,\alpha \mapsto \varnothing \vdash v \sim v : \alpha,$$

which by inversion on LR:VAR is impossible because there is no v such that $v\varnothing v$. Therefore $e[\tau/\alpha]\uparrow$. □

§ Parametricity and type analysis

I now consider the problem of extending the parametricity theorem to all of λ_{SECi}. There are two primary difficulties in doing so.

As an example of the first problem, the following λ_{SECi} term (eliding labels) violates Corollary 2·2·6:

$$\textbf{typecase}\ [\gamma.\textbf{bool}]\ \alpha\ \textbf{true}\ (\Lambda\beta{:}\star.\Lambda\delta{:}\star.\textbf{false})(\Lambda\beta{:}\star.\Lambda\delta{:}\star.\textbf{false}),$$

This expression contradicts confidentiality because substituting **bool** for α and substituting **bool** × **bool** for α will cause the expression to evaluate to different values: **true** versus **false**. It is not possible to directly extend the proof of parametricity to handle **typecase**. The proof would require that the two terms produce related results, even when they may analyze different constructors.

Still, I would like to state properties similar to Corollaries 2·2·6 and 2·2·7 for λ_{SECi}. The problem I describe above can be solved by strengthening the definition of the logical relation. Specifically, by changing the rule LR:ALL to require that τ_1 and τ_2 are β-equivalent:

$$\frac{\forall \tau_1,\tau_2.\tau_1 = \tau_2 : \star, \forall (R \in \tau_1 \leftrightarrow \tau_2).\eta,\alpha \mapsto R \vdash \nu_1[\tau_1] \approx \nu_2[\tau_2] : \sigma \qquad R \text{ consistent}}{\eta \vdash \nu_1 \sim \nu_2 : \forall \alpha{:}\star.\sigma} \text{ LR:ALL-EQ}$$

This revised version of LR:ALL does allow a stronger version of Corollary 2·2·6 to be proven in the presence of **typecase**, but it is so strong that it is vacuous. The example above is resolved simply because the theorem only says anything about the behavior when substituting β-equivalent constructors for α.

This is why tracking information-flow is critical – it allows for a richer definition of equivalence for constructors than β-equivalence. For example, here is the earlier example annotated with information-flow labels:

$$\textbf{typecase}\ [\gamma.(\textbf{bool})\ @\ \top]\ \alpha\ \textbf{true}\ (\Lambda\beta{:}\star^{\top}.\Lambda\delta{:}\star^{\top}.\textbf{false})\ (\Lambda\beta{:}\star^{\top}.\Lambda\delta{:}\star^{\top}.\textbf{false})$$

If α has kind $\star^{\top}$ then as specified by the typing rule WFT:TCASE, the entire expression will have type (**bool**) @ $\top$. As before substituting **bool** × **bool** for α will cause the expression to evaluate to different values: **true** versus **false**. However, in an information-flow type system, equivalence is parametrized by an observer. If the observer is only allowed to observe data with an information content less than $\top$, to that observer **true** and **false** at type (**bool**) @ $\top$ will be indistinguishable. The next section will explain in more detail what it means for constructors to be related in an information-flow kind system.

A second problem that arises when trying to prove a generalization of the parametricity theorem for λ_{SECi} is simply defining the relation. Logical relations are defined inductively over the types of the language. However, in λ_{SECi} the weak-head normal forms of types include (for example) **Typerec** with its scrutinee a variable. It is not obvious what it means for two values to be related at a type like

$$\textbf{Typerec}\ \alpha\ \textbf{bool}\ (\lambda\beta{:}\star^{\perp}\lambda\delta{:}\star^{\perp}.\textbf{bool} \to \textbf{bool})\ (\lambda\beta{:}\star^{\perp}\lambda\delta{:}\star^{\perp}.\textbf{bool} \times \textbf{bool}).$$

The solution that I use, for λ_{SECi}, is to quantify over families of relations between values instead of merely quantifying over relations between values of two specific types. I will explain how this works in more detail when I revisit the logical relation for expressions in § 2·2.

§ Equivalence of constructors

The first step towards a generalized parametricity theorem is formalizing what it means for type constructors to be equivalent in an information-flow kind system. Instead of defining the equivalence inductively over the structure of constructors, like in Appendix B, I define a logical relation between constructors inductively over their kinds.

I write $\tau_1 \approx_\ell \tau_2 : \kappa$ to mean closed constructors τ_1 and τ_2 are related at kind κ with respect to a label, ℓ, called the *observer*. Similarly, the judgment $\nu_1 \sim_\ell \nu_2 : \kappa$ is used to indicate that closed weak-head normal constructors ν_1 and ν_2 are related at kind κ with respect to an observer, ℓ. The grammar of weak-head normal constructors and relations on constructors is defined in Figures 2·9 and 2·10, respectively.

constructor contexts	$\xi ::= \bullet \mid \mathbf{Typerec}\ \xi\ \tau_{\mathbf{bool}}\ \tau_{\rightarrow}\ \tau_{\times} \mid \xi\,\tau$
weak-head normal-form constructors	$\nu ::= \xi\{\alpha\} \mid \mathbf{bool} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \lambda\alpha{:}\kappa.\tau$
weak-head normal-form types	$\zeta ::= (\mathbf{bool})\ @\ \ell \mid (\xi\{\alpha\})\ @\ \ell \mid \sigma_1 \xrightarrow{\ell} \sigma_2 \mid \sigma_1 \times^{\ell} \sigma_2 \mid \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$

Figure 2·9: The grammar of additional syntactic forms in $\lambda_{\mathrm{SEC}i}$.

$$\frac{\ell_1 \not\sqsubseteq \ell_o}{\nu_1 \sim_{\ell_o} \nu_2 : \star^{\ell_1}}\ \textsc{tslr:type-opaq} \qquad \frac{\ell_1 \sqsubseteq \ell_o}{\mathbf{bool} \sim_{\ell_o} \mathbf{bool} : \star^{\ell_1}}\ \textsc{tslr:type-bool}$$

$$\frac{\ell_1 \sqcup \ell_2 \sqsubseteq \ell_3 \qquad \ell_3 \sqsubseteq \ell_o \qquad \tau_1 \approx_{\ell_o} \tau_3 : \star^{\ell_1} \qquad \tau_2 \approx_{\ell_o} \tau_4 : \star^{\ell_2}}{\tau_1 \rightarrow \tau_2 \sim_{\ell_o} \tau_3 \rightarrow \tau_4 : \star^{\ell_3}}\ \textsc{tslr:type-arr}$$

$$\frac{\ell_1 \sqcup \ell_2 \sqsubseteq \ell_3 \qquad \ell_3 \sqsubseteq \ell_o \qquad \tau_1 \approx_{\ell_o} \tau_3 : \star^{\ell_1} \qquad \tau_2 \approx_{\ell_o} \tau_4 : \star^{\ell_2}}{\tau_1 \times \tau_2 \sim_{\ell_o} \tau_3 \times \tau_4 : \star^{\ell_3}}\ \textsc{tslr:type-prod}$$

$$\frac{\forall(\tau_1 \approx_{\ell_o} \tau_2 : \kappa_1).\nu_1\tau_1 \approx_{\ell_o} \nu_2\tau_2 : \kappa_2 \sqcup \ell_1}{\nu_1 \sim_{\ell_o} \nu_2 : \kappa_1 \xrightarrow{\ell_1} \kappa_2}\ \textsc{tslr:arr}$$

$$\frac{\tau_1 \rightsquigarrow^* \nu_1 \qquad \tau_2 \rightsquigarrow^* \nu_2 \qquad \nu_1 \sim_{\ell_o} \nu_2 : \kappa}{\tau_1 \approx_{\ell_o} \tau_2 : \kappa}\ \textsc{tsclr:base}$$

Figure 2·10: Logically related constructors in $\lambda_{\mathrm{SEC}i}$.

Making the distinction between constructors and weak-head normal constructors is especially useful because the head of closed weak-head normal form for constructors will never be **Typerec**.

Constructors that are not in normal form are related by TSCLR:BASE if and only if their weak-head normal forms are related. The rule for type functions, TSLR:ARR, is standard for logical relations.

An anthropomorphic interpretation of the observer is of an individual with the clearance to inspect data with an information content below a specific label in the label lattice. If the observer is an administrator she may be cleared to inspect data with an information content less than $\top$. Guest users of a system might only be allowed to inspect data with an information content of $\bot$. Because such users cannot inspect data with an information content higher than $\bot$, all data with such an information content will appear identical to them. This restriction is enforced by the rule TSLR:TYPE-OPAQ in Figure 2·10. For example, $\mathbf{bool} : \star^{\top}$ and $\mathbf{bool} \times \mathbf{bool} : \star^{\top}$ which carry "high-security" information $\top$, will be indistinguishable to an observer at a "low-security" level $\bot$. Otherwise, the standard equivalence rules TSLR:TYPE-BOOL, TSLR:TYPE-ARR, and TSLR:TYPE-PROD are used.

More formally, the observer label can be understood as a parameter that quotients the logical relation. If the observer is $\top$ then the relation is $\beta\eta$-equivalence of constructors.[3] If the observer is some label, ℓ, less than $\top$, then the relation is $\beta\eta$-equivalence for those constructors with an information content less than or equal to ℓ, and the universal relation for constructors with an information content greater than ℓ.

While the logical relation on constructors was designed so that the it will be the universal relation when the observer is lower than the information content of the constructors, it is not an axiom. Therefore, it is wise to check the definitions by proving the following lemma.

Lemma 2·2·8 (Obliviousness for constructors)**.** *If* $\cdot \vdash \tau_1,\tau_2 : \kappa$ *and* $\mathcal{L}(\kappa) \not\sqsubseteq \ell_o$*, then* $\tau_1 \approx_{\ell_o} \tau_2 : \kappa$*.*

Proof. By simultaneous induction over the structure of $\cdot \vdash \tau_1 : \kappa$ and $\cdot \vdash \tau_2 : \kappa$. □

Another important property of the relation is that it is closed under subsumption. The following lemma verifies the intuition that two related constructors will always stay related when made more restricted.

Lemma 2·2·9 (Constructor relation is closed under subsumption)**.**

If $\kappa_1 \leq \kappa_2$ *and* $\tau_1 \approx_{\ell_o} \tau_2 : \kappa_1$*, then* $\tau_1 \approx_{\ell_o} \tau_2 : \kappa_2$

Proof. By induction over the structure of $\tau_1 \approx_{\ell_o} \tau_2 : \kappa_1$. □

Finally, because I have defined equivalence on constructors in terms of a logical relation, it is useful (and later necessary) to prove a result for type constructors that is similar to parametricity for terms. However, first I must provide a revised definition of what it means for two constructor substitutions to be related. Given,

$$\frac{\forall \alpha{:}\kappa \in \Delta.(\delta_1(\alpha) \approx_{\ell_o} \delta_2(\alpha) : \kappa)}{\delta_1 \approx_{\ell_o} \delta_2 : \Delta}\ \textsc{tsslr:base}$$

the lemma is as follows:

Lemma 2·2·10 (Basic lemma for constructors)**.** *If* $\Delta \vdash \tau : \kappa$ *and* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ *then* $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \kappa$*.*

Proof. By induction over the structure of $\Delta \vdash \tau : \kappa$. See Appendix C·3 for the complete details. □

Now that I have explained how equivalence on constructors is defined for $\lambda_{\text{SEC}i}$, I will examine the revisions necessary to the logical relation on expressions.

§ Related expressions

As with constructors, I parametrize the logical relation on terms by an observer at level ℓ in the label lattice. I write $\eta \vdash e_1 \approx_\ell e_2 : \sigma$ to indicate that terms e_1 and e_2 are related to an observer at level ℓ at type σ, with the relation mapping η. As with constructors, I distinguish between related terms and related

3. The relation is $\beta\eta$-equivalence for type functions, but only β-equivalence for **Typerec**. The reason for this difference is because the logical relation for constructor equivalence is inductively defined on kinds, and because **Typerec** does not introduce a distinguished kind, the only equivalences defined for **Typerec** constructors are given by the rule TSCLR:BASE.

$$\frac{\tau \rightsquigarrow \tau'}{(\tau)\ @\ \ell \rightsquigarrow (\tau')\ @\ \ell}\ \textsc{whr:inj-tc} \qquad \frac{}{(\tau_1 \to \tau_2)\ @\ \ell \rightsquigarrow (\tau_1)\ @\ \ell \xrightarrow{\ell} (\tau_2)\ @\ \ell}\ \textsc{whr:inj-arr}$$

$$\frac{}{(\tau_1 \times \tau_2)\ @\ \ell \rightsquigarrow (\tau_1)\ @\ \ell \times^{\ell} (\tau_2)\ @\ \ell}\ \textsc{whr:inj-prod}$$

Figure 2·11: Type reduction in $\lambda_{\text{SEC}i}$.

$$\frac{\alpha \mapsto R \in \eta \qquad (\ell_1 \sqsubseteq \ell_o) \Longrightarrow (v_1 R_\xi^{\ell_1} v_2)}{\eta \vdash v_1 \sim_{\ell_o} v_2 : (\xi\{\alpha\})\ @\ \ell_1}\ \textsc{slr:con} \qquad \frac{(\ell_1 \sqsubseteq \ell_o) \Longrightarrow (v_1 = v_2)}{\eta \vdash v_1 \sim_{\ell_o} v_2 : (\mathbf{bool})\ @\ \ell_1}\ \textsc{slr:bool}$$

$$\frac{\forall(\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma_1).\eta \vdash v_1 e_1 \approx_{\ell_o} v_2 e_2 : \sigma_2 \sqcup \ell_1}{\eta \vdash v_1 \sim_{\ell_o} v_2 : \sigma_1 \xrightarrow{\ell_1} \sigma_2}\ \textsc{slr:arr}$$

$$\frac{\eta \vdash \mathbf{fst}\ v_1 \approx_{\ell_o} \mathbf{fst}\ v_2 : \sigma_1 \sqcup \ell_1 \qquad \eta \vdash \mathbf{snd}\ v_1 \approx_{\ell_o} \mathbf{snd}\ v_2 : \sigma_2 \sqcup \ell_1}{\eta \vdash v_1 \sim_{\ell_o} v_2 : \sigma_1 \times^{\ell_1} \sigma_2}\ \textsc{slr:prod}$$

$$\frac{\begin{array}{c}\forall(\tau_1 \approx_{\ell_o} \tau_2 : \star^{\ell_2}).\forall(R_\xi^{\ell_2} \in \delta_1((\xi\{\tau_1\})\ @\ \ell_2) \leftrightarrow \delta_2((\xi\{\tau_2\})\ @\ \ell_2)).\\ \eta,\alpha \mapsto R \vdash v_1[\tau_1] \approx_{\ell_o} v_2[\tau_2] : \sigma \sqcup \ell_1 \qquad R \text{ consistent}\end{array}}{\eta \vdash v_1 \sim_{\ell_o} v_2 : \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma}\ \textsc{slr:all}$$

$$\frac{e_1 \rightsquigarrow^* v_1 \qquad e_2 \rightsquigarrow^* v_2 \qquad \sigma \rightsquigarrow^* \zeta \qquad \eta \vdash v_1 \sim_{\ell_o} v_2 : \zeta}{\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma}\ \textsc{slr:term} \qquad \frac{e_1 \uparrow}{\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma}\ \textsc{slr:divr1}$$

$$\frac{e_2 \uparrow}{\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma}\ \textsc{slr:divr2}$$

Figure 2·12: Logically related terms in $\lambda_{\text{SEC}i}$.

normal forms, writing the judgment $\eta \vdash v_1 \sim_\ell v_2 : \zeta$ to indicate that values v_1 and v_2 are related to an observer at level ℓ at the weak-head normal type ζ, with the relation mapping η. These relations, as defined in Figure 2·12, are similar to the ones in Figure 2·6. One difference is that I only relate values at weak-head normal types ζ, defined in Figure 2·9.

Restricting the value relation to weak-head normal types makes the logical relation much easier to state and understand. For example, the term $\langle \mathbf{true}, \mathbf{false} \rangle$ is well typed with the equivalent types $(\mathbf{bool} \times \mathbf{bool})\ @\ \ell$ and $(\mathbf{bool})\ @\ \ell \times^{\ell} (\mathbf{bool})\ @\ \ell$. However, restricting the relation to weak-head normal types means that only the case for $(\mathbf{bool})\ @\ \ell \times^{\ell} (\mathbf{bool})\ @\ \ell$ must be considered in the inductive proof.

Like constructors, the relation over terms is defined so that terms with a greater information content than the observer will be indistinguishable. This is enforced by the precondition $\ell_1 \sqsubseteq \ell_o$ found in SLR:CON and SLR:BOOL. The antecedent relations in SLR:ALL, SLR:ARR, and SLR:PROD all have their types joined with ℓ_1;

this accounts for information gained by destructing the value. The following lemma verifies the intuitions concerning indistinguishability:

Lemma 2·2·11 (Obliviousness for terms)**.** *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *and* $\mathcal{L}(\zeta) \not\sqsubseteq \ell_o$ *and*

- $\Delta^\star;\cdot \vdash \nu_1,\nu_2 : \zeta$ *then* $\eta \vdash \delta_1(\nu_1) \sim_{\ell_o} \delta_2(\nu_2) : \zeta$.
- $\Delta^\star;\cdot \vdash e_1,e_2 : \sigma$ *then* $\eta \vdash \delta_1(e_1) \approx_{\ell_o} \delta_2(e_2) : \sigma$.

Proof. The first part follows from induction on ζ and the second part from Theorem 2·1·2 (Type safety). □

There are two other significant differences between Figures 2·6 and 2·12: additional preconditions in SLR:ALL, and generalizing LR:VAR to SLR:CON. The rule SLR:CON solves the problem with **Typerec** appearing in the weak-head normal form of types. It generalizes LR:VAR to terms related at a constructor that cannot be normalized further because of an undetermined type variable. I characterize these constructors with constructor contexts, ξ, defined in Figure 2·9. Contexts are holes $\bullet$, **Typerec**s of a context, or a context applied to an arbitrary constructor. I write $\xi\{\tau\}$ for filling a context's hole with τ.

Previously, values were related at a type variable only if they were in the relation mapped to that variable by η. Here η maps to families of relations. I write R^ℓ_ξ for the application of R to a label ℓ and a context ξ, yielding a relation. Therefore, when I write

$$R^\ell_\xi \in \delta_1((\xi\{\tau_1\}) \mathbin{@} \ell) \leftrightarrow \delta_2((\xi\{\tau_2\}) \mathbin{@} \ell),$$

I mean that R is a dependent function of ℓ and ξ yielding a relation on values of type $\delta_1((\xi\{\tau_1\}) \mathbin{@} \ell)$ and $\delta_2((\xi\{\tau_2\}) \mathbin{@} \ell)$.

This move from relations to families of relations makes it more difficult to use the resulting generalized parametricity theorem. This is primarily because in standard parametricity it is only necessary to choose a relationship between values of two fixed types, while in generalized parametricity it is necessary to choose a family of relationship between values of arbitrary type. This is because the constructor context, ξ, determines the types of the values R^ℓ_ξ must relate.

To date I have been unable to devise any non-trivial families of relations that are not parametric in their constructor context. It is open question whether there are interesting families of relations that are not parametric in their constructor context. Because constructor contexts were introduced to handle **Typerec**, if it were removed from λ_{SEC^i} this problem would go away. There may be less drastic solutions and I will discuss some of my ideas in § 6·1. Fortunately, the families of relations used to prove the confidentiality and integrity corollaries, the universal relation and the null relation, respectively, are parametric in their constructor context.

As with standard parametricity, quantification over R is required to be consistent. In addition to being closed under erasure of fixed point annotations, as I described for the vanilla parametricity theorem in § 2·2, relations are required to be closed under subtyping. That means if $\nu_1 R^{\ell_1}_\xi \nu_2$ and $\ell_1 \sqsubseteq \ell_2$ then it must also be the case that $\nu_1 R^{\ell_2}_\xi \nu_2$.

It is important that the logical relation itself is consistent, that is, closed under subsumption and erasure.

Lemma 2·2·12 (Term relation is consistent)**.**

- *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and*
  $\Delta^\star \vdash \sigma_1 \preceq \sigma_2$ *and*
  $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma_1$ *then*
  $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma_2$.

- *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and*
  $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma$ *and*
  $e_1 \preceq e_1'$ *and* $e_2 \preceq e_2'$ *then*
  $\eta \vdash e_1' \approx_{\ell_o} e_2' : \sigma$.

Proof. For the first part, straightforward induction over the structure of σ_1 and for the second part, straightforward induction over the structure of $\eta \vdash e_1' \approx_{\ell_o} e_2' : \sigma$. □

I write $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ to mean that the mapping η is well-formed with respect to a pair of type substitutions, δ_1 and δ_2, as defined in the rule:

$$\frac{\forall \alpha{:}\star^{\ell_1} \in \Delta^\star.(\eta(\alpha)^{\ell_1}_{\xi} \in \delta_1((\xi,\{\alpha\}) \mathbin{@} \ell_1) \leftrightarrow \delta_2((\xi,\{\alpha\}) \mathbin{@} \ell_1)) \qquad \eta(\alpha) \text{ consistent}}{\delta_1,\delta_2 \vdash \eta : \Delta^\star} \textsc{relm:reg}$$

The last significant difference in Figure 2·6 is that LR:DIVR has been split into SLR:DIVR1 and SLR:DIVR2. Terms in $\lambda_{\text{SEC}i}$ are related if either diverges, as opposed to my earlier definition where divergent terms were only related to other divergent terms. At first, this change might seem like a significant weakening of the relation. In particular, the logical relation is no longer transitive. However, this definition is standard for information-flow logical relations proofs with recursion (Abadi et al. 1999; Zdancewic 2002). I will discuss how this requirement is merely an artifact of call-by-value information-flow in the next subsection.

§ Generalized parametricity

Before stating the generalized parametricity theorem, the notion of related term substitutions must be defined. Given related type substitutions, $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$, and a well-formed mapping, $\delta_1,\delta_2 \vdash \eta : \Delta^\star$, term substitutions are related if they map variables to related terms.

$$\frac{\forall x{:}\sigma \in \Gamma.(\eta \vdash \gamma_1(x) \approx_{\ell_o} \gamma_2(x) : \sigma)}{\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma} \textsc{sslr:base}$$

The only change from SLR:BASE is the additional of a label ℓ_o for the observer.

Theorem 2·2·13 (Generalized parametricity)**.** *If* $\Delta^\star;\Gamma \vdash e : \sigma$ *and*
$\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *and*
$\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and*
$\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ *then*
$\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma$.

Kinds

$$\begin{array}{rcl} [\![\star]\!] & \coloneqq & \star^{\top} \\ [\![\kappa_1 \to \kappa_2]\!] & \coloneqq & [\![\kappa_1]\!] \xrightarrow{\top} [\![\kappa_2]\!] \\ \\ [\![\mathbf{bool}]\!] & \coloneqq & (\mathbf{bool}) \mathbin{@} \bot \end{array}$$

Types

$$\begin{array}{rcl} [\![\sigma_1 \times \sigma_2]\!] & \coloneqq & [\![\sigma_1]\!] \times^{\bot} [\![\sigma_2]\!] \\ [\![\sigma_1 \to \sigma_2]\!] & \coloneqq & [\![\sigma_1]\!] \xrightarrow{\bot} [\![\sigma_2]\!] \\ [\![\forall \alpha{:}\kappa.\sigma]\!] & \coloneqq & \forall^{\bot} \alpha{:}[\![\kappa]\!].[\![\sigma]\!] \end{array}$$

Expressions

$$\begin{array}{rcl} [\![\mathbf{true}]\!] & \coloneqq & \mathbf{true} \\ [\![\mathbf{false}]\!] & \coloneqq & \mathbf{false} \\ [\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] & \coloneqq & \mathbf{if}\ \langle [\![e_1]\!] \rangle\ \mathbf{then}\ [\![e_2]\!]\ \mathbf{else}\ [\![e_3]\!] \\ [\![\langle e_1, e_2 \rangle]\!] & \coloneqq & \langle [\![e_1]\!], [\![e_2]\!] \rangle \\ [\![\mathbf{fst}\, e]\!] & \coloneqq & \mathbf{fst}\, [\![e]\!] \\ [\![\mathbf{snd}\, e]\!] & \coloneqq & \mathbf{snd}\, [\![e]\!] \\ [\![\lambda x{:}\sigma.e]\!] & \coloneqq & \lambda x{:}[\![\sigma]\!].[\![e]\!] \\ [\![e_1 e_2]\!] & \coloneqq & [\![e_1]\!][\![e_2]\!] \\ [\![\Lambda \alpha{:}\kappa.e]\!] & \coloneqq & \Lambda \alpha{:}[\![\kappa]\!].[\![e]\!] \\ [\![e[\tau]]\!] & \coloneqq & [\![e]\!][\![\tau]\!] \end{array}$$

Relations

$$\begin{array}{rcl} [\![\cdot]\!] & \coloneqq & \cdot \\ [\![\eta, \alpha \mapsto R]\!] & \coloneqq & [\![\eta]\!], \alpha \mapsto \begin{cases} \{(v_1, v_2) \mid v_1\ R\ v_2, (\cdot; \cdot \vdash v_1 : \tau_1), (\cdot; \cdot \vdash v_2 : \tau_2)\} & \xi = \bullet \text{ and } \ell = \bot \\ \{(v_1, v_2) \mid (\cdot; \cdot \vdash v_1 : \tau_1), (\cdot; \cdot \vdash v_2 : \tau_2)\} & \xi = \bullet \text{ and } \ell \neq \bot \\ \varnothing & \text{otherwise} \end{cases} \\ & & \text{where } R \in \tau_1 \leftrightarrow \tau_2 \end{array}$$

Figure 2·13: The encoding for standard parametricity.

Proof. As with standard parametricity, the proof is by induction over $\Delta^{\star}; \Gamma \vdash e : \sigma$. In addition to the lemmas mentioned in § 2·2 and § 2·2, Lemma 2·2·5 must be extended in the straightforward manner. See Appendix C·3 for the complete details. □

I call this theorem generalized parametricity because I conjecture that Theorem 2·2·1 can be recovered via an encoding:

- Restrict the label lattice to two elements, $\bot$ and $\top$ where $\bot \sqsubseteq \top$.
- For every kind κ in $\Delta^{\star}$, Γ, e, and σ require $\mathcal{L}(\kappa) = \top$.
- For every type σ' in Γ, e, and σ require $\mathcal{L}(\sigma') = \bot$.
- Require that the observer be $\bot$.

Figure 2·13 makes this encoding explicit, allowing the relationship between standard parametricity and generalized parametricity to be described formally.

Conjecture 2·2·14 (Generalized parametricity subsumes standard parametricity)**.**
If $\Delta^\star;\Gamma \vdash e : \sigma$ *and*
$\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ *and*
$\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma$*, then*
$\eta \vdash \delta_1(\gamma_1(e)) \approx \delta_2(\gamma_2(e)) : \sigma$ *iff* $[\![\eta]\!] \vdash [\![\delta_1(\gamma_1(e))]\!] \approx_\bot [\![\delta_2(\gamma_2(e))]\!] : [\![\sigma]\!]$
where $\delta_1(\gamma_1(e)) \uparrow$ *iff* $\delta_2(\gamma_2(e)) \uparrow$.

I expect that the proof will follow by induction over the structure of the polymorphic λ-calculus typing judgment, $\Delta^\star;\Gamma \vdash e : \sigma$.

However, this encoding is not perfect because LR:DIVR has been split into a disjunction with the rules SLR:DIVR1 and SLR:DIVR2. Therefore, Theorem 2·2·13 makes a weaker claim about the termination behavior of related terms than Theorem 2·2·1. This difference is accounted for in Conjecture 2·2·14 by the side condition $\delta_1(\gamma_1(e)) \uparrow$ iff $\delta_2(\gamma_2(e)) \uparrow$. Furthermore, the difference in how the theorems treat non-termination does impact my results – consider the generalized version of Corollary 2·2·6:

Corollary 2·2·15 (Confidentiality)**.** *If* $\alpha{:}\star^\top; x{:}(\alpha) @ \bot \vdash e : (\mathbf{bool}) @ \bot$ *then for any* $\cdot \vdash v_1 : \tau_1$ *and* $\vdash v_2 : \tau_2$ *if* $e[\tau_1/\alpha][v_2/x]$ *and* $e[\tau_2/\alpha][v_2/x]$ *both terminate, they will produce the same value.*

Proof. The details of the proof are very similar to those for Corollary 2·2·6. Full details can be found in § C·3. □

This corollary states that what is substituted for α and x will not affect the value computed by e. However, it is possible that the choice of α and x could cause e to diverge. What is happening?

Unlike standard parametricity, Theorem 2·2·13 has an explicit observer. Standard parametricity has an implicit observer that can observe all computations. What makes information-flow techniques work is that some computations are opaque to the observer. Furthermore, the results of these computations are also inaccessible to the observer, making them effectively dead code. However, because the operational semantics I chose to use for λ_{SEC_i} is call-by-value, dead code must be executed even though the result is never used.

For example, the following expression is well-typed in λ_{SEC_i} with type $\mathbf{bool}^\bot$ under the assumption α has kind $\star^\top$:

$$\begin{aligned}(\lambda x{:}(\mathbf{bool}) @ \top.\mathbf{true})\ (&\mathbf{typecase}[\gamma.(\mathbf{bool}) @ \top]\alpha\\ &(\mathbf{true})\\ &(\Lambda\beta{:}\star^\top.\Lambda\delta{:}\star^\top.\mathbf{fix}\ y{:}(\mathbf{bool}) @ \top.y)\\ &(\Lambda\beta{:}\star^\top.\Lambda\delta{:}\star^\top.\mathbf{false}))\end{aligned}$$

Corollary 2·2·15 states that if two related constructors are substituted for the free type variable α, in the expression above, that the two resulting expressions will be related. If **bool** is substituted for α the expression will evaluate to **true**, but if $\mathbf{bool} \to \mathbf{bool}$ is substituted for α then the expression will diverge. Therefore, because one of the expressions diverges, the corollary has not been contradicted.

However, note that the expression

$$(\mathbf{typecase}[\gamma.(\mathbf{bool}) @ \top]\alpha\ (\mathbf{true})(\Lambda\beta{:}\star^\top.\Lambda\delta{:}\star^\top.\mathbf{fix}\ y{:}(\mathbf{bool}) @ \top.y)(\Lambda\beta{:}\star^\top.\Lambda\delta{:}\star^\top.\mathbf{false})),$$

is completely dead code because when it does evaluate to a value, it is simply thrown away. If λ_{SECi} is given a call-by-name operational semantics, the original expression above is operationally equivalent to the expression **true**. I conjecture that all such discrepancies in termination behavior are a result of dead code. Therefore, by using a call-by-name operational semantics, an exact correspondence between standard parametricity and generalized parametricity could be recovered.[4]

§ Applications of generalized parametricity

A typical corollary of Theorem 2·2·13 is normally called noninterference; the property that it is possible to substitute values indistinguishable to the present observer and get indistinguishable results.

Corollary 2·2·16 (Noninterference)**.** *If* $\cdot, x{:}\sigma_1 \vdash e : \sigma_2$ *where* $\mathcal{L}(\sigma_1) \not\sqsupseteq \mathcal{L}(\sigma_2)$*, then for any* $\vdash v_1 : \sigma_1$ *and* $\vdash v_2 : \sigma_1$ *it is the case that if both* $e[v_1/x]$ *and* $e[v_2/x]$ *terminate, they will both produce the same value.*

Proof. Proceeds in a similar fashion to Corollary 2·2·15. □

More importantly, it is also possible to restate the corollaries of standard parametricity proven earlier. The previous subsection stated the revised corollary for confidentiality. The same can be done for integrity:

Corollary 2·2·17 (Integrity)**.** *If* $\alpha{:}\star^{\top}; \cdot \vdash e : (\alpha)\ @\ \bot$ *then* $e[\tau/\alpha]$ *for any* τ *must diverge.*

Proof. The details of the proof are very similar to those for Corollary 2·2·7. Full details can be found in Appendix C·3. □

While these corollaries are very similar in spirit to the ones derived from standard parametricity, it is possible to make much richer and more refined claims because the label lattice expands upon the implicit two level lattice used by parametricity. For example, it is possible to label each abstract data type with a distinct label. This makes it possible to understand which abstract types depend upon each other; the fact that all abstract types in standard parametricity are labeled with $\top$ means that it is not possible to discern their interdependenices. Furthermore, using distinct labels makes it possible to reason about the abstraction properties of each data type separately. I will explore this possibility in greater detail in § 4.

§ 2·3 Related work

The design of λ_{SECi} and the proof of generalized parametricity draws heavily upon previous work on type analysis, parametricity, and information flow.

Most information flow systems use a lattice model originating from work by Bell and La Padula (1975) and Denning (1976). The earliest work on static information flow dates back to Denning and Denning (1977). Volpano, Smith, and Irvine (1996) showed that Denning's work could be formulated as a type system and proved its soundness with respect to noninterference. Heintze and Riecke (1998)

4. The only part of the proof for Theorem 2·2·13 that would need to change is the proof of obliviousness for terms, Lemma 2·2·11.

formalized information-flow and integrity in a typed λ-calculus with references, the SLam calculus, and proved a number of soundness and noninterference results. Pottier and Simonet (2003) have developed an extension of ML, called FlowCaml, and have shown noninterference using an alternative syntactic technique.

Prior to this research, FlowCaml was the only language with polymorphism and a noninterference proof. However, FlowCaml does not have any mechanisms for TDP and can rely on standard parametricity for types. There was some prior research on noninterference with principal polymorphism by Tse and Zdancewic (2004A), and later concurrently with this research they investigated a language with type polymorphism where labels and principals were integrated into the language of types (2005). Furthermore, because their goal was to support runtime decisions based upon principals, and because principals in their formalization are a special form of type, their language provides a form of runtime type analysis. However, their noninterference theorems focus on how related terms affect computation and do not consider how related types would alter computations.

While research into abstraction properties predates his work, Reynolds (1974; 1983) was the first to show how the parametricity theorem could be used to prove properties about representation independence in the polymorphic λ-calculus. Reynold's proofs were for a polymorphic λ-calculus without higher-kinded types. While I have restricted $\lambda_{\mathrm{SEC}i}$ to disallow polymorphic functions over higher-kinded types, most of the machinery necessary to handle higher-kinded types has been developed because type operators are allowed to abstract over higher-kinded type variables. Girard, in his dissertation (1972), did present a form of logical relation for the λ-calculus without higher-kinded types.[5]

Gallier (1990) later gave a detailed survey of variations on formalizing what Girard called the method of "Candidats de Reductibilité", including the extensions to higher-kinds. However Gallier focused on strong normalization, so he only studied a unary logical relation. Kučan, in his dissertation (2007), did consider an interpretation for the λ-calculus without higher-kinded types that extended to n-ary relations, but his interpretation is untyped.

Finally, following the publication of my original work on generalized parametricity, Vytiniotis and Weirich (2007B) developed a detailed formalization of parametricity for the higher-order polymorphic λ-calculus. However, instead of building their formalization around the canonical forms of types, as I have done, they require an additional consistency requirement that their relations must behave the same on β-equivalent types.

My generalized parametricity result for $\lambda_{\mathrm{SEC}i}$ directly builds upon the methods of Zdancewic (2002) and Pitts (2005). Other researchers have noticed the connection between parametricity and noninterference. For example, the work of Tse and Zdancewic (2004B) compliments my research by showing how parametricity can be used to prove noninterference. Tse and Zdancewic do so by encoding Abadíet al.'s (1999) dependency core calculus into the polymorphic λ-calculus.

The fact that runtime type analysis (and other forms of ad-hoc polymorphism) breaks parametricity has been long understood, but little has been done to reconcile the two. Leifer et al. (2003) design a system that preserves type abstraction in the presence of (un)marshalling. This is a weaker result because marshalling is merely a single instance of an operation using run-time type analysis. Rossberg (2003) and

5. However, it is not clear whether his relation was more than unary. I have not yet attempted to study his dissertation in detail because it is written in Français.

Vytiniotis, Washburn, and Weirich (2005) use generative types to hide type information in the presence of run-time analysis, relying on colored-brackets (Grossman, Morrisett, and Zdancewic 2000) to provide easy access. However, none of this work has formalized the abstraction properties that their systems provide.

Finally, following the original publication of the work on generalizing parametricity, Vytiniotis and Weirich (2007A; 2007B) have investigated a more traditional parametricity result for a language with type representations in the style of λ_R. Their work is the most closely related to the research on generalized parametricity.

Their initial work (2007A) does not handle type operators and type analysis is based upon type representations. There are three significant differences between the language they studied in that work and $\lambda_{\mathrm{SEC}i}$.

The first difference is that they provide a special "top" type representation called $\mathsf{R}_{\mathtt{any}}$. They can use this representation to prove properties that have no correspondence in generalized parametricity as stated here. If $\mathsf{R}_{\mathtt{any}}$ is omitted from their language, the properties that can be proven in their language are a subset of those that can be derived from generalized parametricity. Using $\mathsf{R}_{\mathtt{any}}$ as an argument to a type analyzing function is a way of forcing functions to behave parametrically at runtime. It is possible to label programs in $\lambda_{\mathrm{SEC}i}$ to force functions to behave parametrically statically, but there is no dynamic analog.

The second difference is that their language allows impredicative rather than predicative type quantification. Therefore, it is possible to write programs and free theorems about polymorphic functions that can be instantiated with polymorphic types themselves. However, because they do not provide a type representation for polymorphic types, there is no interesting interaction between type analysis and polymorphic types just as in $\lambda_{\mathrm{SEC}i}$. The primary obstacle to allowing impredicative type quantification in $\lambda_{\mathrm{SEC}i}$ comes from **Typerec**. Naïve extensions for analyzing higher-order types at the level of types rather than terms will make type equality undecidable. Extending $\lambda_{\mathrm{SEC}i}$ with a top type would be one way to allow impredicative quantification and avoid this problem.

The third difference is that because type representations are required to perform type analysis, it is possible to completely prevent type analysis by simply not providing a corresponding representation for an abstract type. As I discussed in § 1·4, using type representations in this fashion is a form of access control. However, I conjecture that nearly all free theorems that can be derived by withholding representations can be emulated in $\lambda_{\mathrm{SEC}i}$ with appropriate labeling. For example, the type $\forall^{\bot}\alpha{:}\star^{\bot}.(\alpha)\,@\,\bot \xrightarrow{\bot} (\alpha)\,@\,\bot$ should have similar inhabitants to the type $\forall\alpha.\mathsf{R}[\alpha] \to \alpha \to \alpha$ in their language. Correspondingly, a function with the type $\forall\alpha.\alpha \to \alpha$ in their language should have similar inhabitants to the type $\forall^{\bot}\alpha{:}\star^{\top}.(\alpha)\,@\,\bot \xrightarrow{\bot} (\alpha)\,@\,\bot$ in $\lambda_{\mathrm{SEC}i}$ (modulo the termination discrepancy described in § 2·2).

The more recent work by Vytiniotis and Weirich (2007B) on the language R_ω does address type-operators, as described above in my discussion of higher-order parametricity, but does not examine the problems that arise from including type-level type analysis. Again they make use of type representations, but do not include the $\mathsf{R}_{\mathtt{any}}$ type representation. Unlike their prior work, in R_ω it is possible to prove interesting results, that have no analog in $\lambda_{\mathrm{SEC}i}$, about the static behavior of programs that use type analysis. Again there are three significant differences between R_ω and $\lambda_{\mathrm{SEC}i}$. The first two differences are impredicative polymorphism and the use of type representations for access control, which I discussed

earlier. The third significant difference arises because they allow quantification over higher-kinded types. Their central result is a proof of partial correctness for *generic* type-safe cast from the free theorem for its type. In $\lambda_{\mathrm{SEC}i}$, a type-safe cast can be written and has the type

$$\forall^{\bot}\alpha{:}\star^{\bot}.\forall^{\bot}\beta{:}\star^{\bot}.(\alpha)\ @\ \bot \xrightarrow{\bot} ((\beta)\ @\ \bot +^{\bot} 1).$$

In R_{ω} a generic type-safe cast quantifies over a type-operator and has the type

$$\forall\delta{:}\star \to \star.\forall\alpha{:}\star.\forall\beta{:}\star.\mathsf{R}[\alpha] \to \mathsf{R}[\beta] \to \delta\alpha \to (\delta\beta) + 1.$$

Their parametricity theorem can be used to derive that any implementation of this type, if it returns a value of type $\delta\beta$, that value will be identical to the input value with type $\delta\alpha$. That is, a generic type-safe cast cannot subtly modify the input based upon its representation.

However, Vytiniotis and Weirich's (2007A; 2007B) results are based upon the use of type representations, which, as I described in § 1·4, is a form of dynamic access control. Because access control mechanisms cannot capture dependencies, they cannot be used to prove results about confidentiality and integrity independently, like can be done using generalized parametricity. Furthermore, type representations are values; there is no mechanism in their language to reason about the dependencies between abstract types. Finally, the use of an explicit lattice in $\lambda_{\mathrm{SEC}i}$ allows for cleanly reasoning about the confidentiality and integrity of several ADTs simultaneously, along with the relationships between them.

3

Programming with types in InforML

> This dissertation is about defining operations with types.
>
> Stephanie Weirich (*Programming with Types*, 2002)

In the previous chapter, I developed generalized parametricity as a foundational theory for reasoning about abstract data types in the presence of runtime type analysis. In this chapter, I show how these ideas can be realized in a practical programming language called InforML. InforML is a member of the ML family of languages, extended with primitives for reflecting on type meta-data and an information-flow type and kind system.

I will begin by reviewing how type-directed programming in the core of InforML differs from $\lambda_{\text{SEC}i}$. After introducing the differences in the languages, I will move on to explaining the language features in InforML that have no analog in $\lambda_{\text{SEC}i}$: modules, generative types, and dynamic information-flow. I will conclude my introduction to programming in InforML with an example combining all of these features, and then discuss InforML's relationship to other programming languages.

I will assume familiarity with ML-like languages and I will focus mainly on the novel aspects of InforML. The complete grammar for InforML can be found in § D.

§ 3·1 The basics of InforML

This section will explain the semantic and syntactic differences between $\lambda_{\text{SEC}i}$ and InforML's core language:

- label, higher-order type, and constrained polymorphism,
- local type inference,
- type patterns,

- types and type constructors are combined,
- and the program counter label.

To illustrate these differences, I will use as a running example a type-directed function, "to string", for converting data to human-readable strings. I will begin with an overview of the differences before addressing some points in greater detail in the coming subsections. At the end of this section, I will return to the InforML implementation of "to string" to review how it works and how it typechecks.

Assuming an extension of λ_{SECi} with a string type constructor, **string**, with kind $\star^{\bot}$ and an infix string concatenation function (^) with type (**string**) @ $\ell \xrightarrow{\bot}$ (**string**) @ $\ell \xrightarrow{\bot}$ (**string**) @ ℓ, for some predetermined label ℓ, an implementation of "to string" in λ_{SECi}, might look like:

$$
\begin{array}{l}
\textbf{fix}\ \mathtt{toString}{:}(\forall^{\bot}\alpha{:}\star^{\ell}.(\alpha)\ @\ \ell \xrightarrow{\bot} (\textbf{string})\ @\ \ell).\Lambda\alpha{:}\star^{\ell}. \\
\quad \textbf{typecase}[\delta.\textbf{string}]\ \alpha \\
\quad\quad (\lambda arg{:}\textbf{bool}.\textbf{if}\ arg\ \textbf{then "True" else "False"}) \\
\quad\quad (\Lambda\beta{:}\star^{\ell}.\Lambda\gamma{:}\star^{\ell}.\lambda arg{:}(\beta \to \gamma)\ @\ \ell.\text{"<}\textbf{Function}\text{>"}) \\
\quad\quad (\Lambda\beta{:}\star^{\ell}.\Lambda\gamma{:}\star^{\ell}.\lambda arg{:}(\beta \times \gamma)\ @\ \ell. \\
\quad\quad\quad \textbf{"("} \hat{}\ (\mathtt{toString}[\beta](\textbf{fst}\ arg)) \hat{}\ \textbf{", "} \hat{}\ (\mathtt{toString}[\gamma](\textbf{snd}\ arg)) \hat{}\ \textbf{")"})
\end{array}
$$

In the function toString, if **typecase** determines that α is of type **bool** it returns a function that uses a conditional to choose the appropriate string for arg. In the case that α is a function type, toString simply returns a constant function returning string `"<Function>"`, as there is no way to further inspect a functional value. Finally, in the case that α is a tuple, toString returns a function that will invoke toString recursively on the first and second projections of the tuple and the results are concatenated together.

Each of **typecase**'s branches just returns a **string**, so its type does not depend upon the type of the scrutinee. Consequently, the annotation [δ.**string**], which is used specify how the type of the overall **typecase** expression depends upon the scrutinee, does not need to make use of δ. Unfortunately, in λ_{SECi}, any implementation of toString is restricted to only work on data labeled with a predetermined label ℓ.

Below, I have rewritten the toString function in InforML. An abbreviated grammar for InforML can be found in Figure 3·1.

```
fun toString : ∀⟨l:Lab|α: * @ l|(info α) = l⟩ α -(l|⊥)-> String @ l
fun toString ⟨l|α⟩ arg =
    typecase α
           | Bool @ l      =>
             if arg then "True" else "False" end
           | _ -(_ | _)-> _ =>
             "<Function>"
           | (β, ψ)         =>
             "(" ^ (toString ⟨l|β⟩ (arg.0)) ^ "," ^ (toString ⟨l|ψ⟩ (arg.1)) ^ ")"
         end
```

This example illustrates all five differences between λ_{SECi} and core InforML. However, this implementation of the `toString` function is far too simplistic for practical use – it only handles a fixed subset of values

variances	π	::=	`+`	covariant
		|	`-`	contravariant
		|	`±`	invariant
kinds	κ	::=	`*@ Л`	type classifiers
		|	`Lab -(π)-> κ`	label functions
		|	κ_1 `-(π)->` κ_2	type functions
atomic labels	ℓ	::=	`l`	label variables
		|	`(⊥ | ⊤)`	bottom and top labels
full labels	Л	::=	ℓ	atomic labels
		|	`info τ`	information content of τ
		|	$Л_1$`⊔ … ⊔` $Л_n$	label join
constraints	C	::=	$Л_1$ `(<: | = | :>)` $Л_2$	label comparison
		|	C_1 `&` … `&` C_n	conjunction
polytypes	σ	::=	`∀(([l:Lab)*|(α:κ)*|C?)` σ	universal quantification
		|	τ	monotypes
monotypes	τ	::=	α	variables
		|	`(Int | String | Bool)`	primitives
		|	τ_1 `-(`ℓ_1`|`ℓ_2`)->` τ_2	term functions
		|	`λ(l:Lab | α:κ)=(π)=>` τ	type functions
		|	$\tau_1\tau_2$	type application
		|	τ `@` ℓ	label application
		|	`(`τ_1`, … ,`τ_n`)`	tuples
label patterns	lp	::=	`_`	wildcard
		|	ℓ	atomic labels
type patterns	φ	::=	`_`	wildcard
		|	α	type variable
		|	`(Int | String | Bool)`	primitives
		|	φ_1 φ_2	type application
		|	φ `@ lp`	label application
		|	φ_1 `-(`lp_1`|`lp_2`)->` φ_2	term function
		|	`(`φ_1`, … ,` φ_n`)`	tuples
term patterns	p	::=	`_`	wildcard
		|	x	variable binding
		|	`(True | False | i | "strings")`	values
		|	`(`p_1`, … ,` p_n`)`	tuples
term matches	u	::=	`p => e | p =(`ℓ`)=> e`	
type matches	μ	::=	φ `=> e`	
expressions	e	::=	x`(⟨`Л*`|`τ*`⟩)?`	instantiation
		|	`(True | False | i | "strings")`	values
		|	`(`e_1`, … ,` e_n`)`	tuples
		|	`e.n`	tuple projection
		|	e_1 `(andalso | orelse)` e_2	short-circuiting “and” and “or”
		|	`λ(|)?` u_1 `| … |` u_n `end`	anonymous functions
		|	`let ld* in e end`	let expression
		|	`if` e_1 `then` e_2 `else` e_3 `end`	conditional
		|	`case e(of | |)` u_1 `| … |` u_n `end`	term case
		|	`typecase` τ `(of | |)` μ_1 `| … |` μ_n `end`	type case
		|	e_1e_2	term application
function binding	fb	::=	x`(⟨`l*`|`α*`⟩)?(`p`)+(: σ)? = e`	
local declarations	ld	::=	`fun` fb_1 `and … and` fb_n	recursive function
		|	`fun x : σ`	type annotation
		|	`val p = e`	“let”

Figure 3·1: The abbreviated grammar of InforML.

possible in InforML. Even though it is not a realistic implementation, it is still a useful point of comparison between $\lambda_{\text{SEC}i}$ and InforML. I will explain how to address the limitations of this implementation of `toString` later in this chapter.

All functions in InforML are required to be preceded by a type signature, as shown at the beginning of the InforML version of `toString`. This is because InforML, unlike most members of the ML family of languages, uses local type inference (Pierce and Turner 2000) rather than global type inference (Damas and Milner 1982). Local type inference works by a combination of bidirectional typechecking and synthesizing instantiations for polymorphic functions from their arguments. When writing examples, I may sometimes omit a function's type signature if I have already given it earlier in the text or if it is clear from the example what it should be.

The signature above states that `toString` has the type

```
∀⟨l:Lab|α:* @ l|(info α) = l⟩ α -(l|⊥)-> String @ l.
```

The `∀` specifies that this type is a universally quantified type, with the variables it binds enclosed within the angle brackets `⟨ … ⟩`. I will often describe this pair of angle brackets and their contents as the *quantifier block*.

The first part of the quantifier block, `l:Lab`, says that this function quantifies over the label `l`. As mentioned above, InforML includes label polymorphism. Label polymorphism is very important to writing reusable programs in an information-flow type system. For example, the the $\lambda_{\text{SEC}i}$ version of `toString` could only operate on inputs labeled with a predetermined label ℓ.

Inside the quantifier block, the list of quantified label variables must always precede the list of quantified type variables, separated by a vertical bar (`|`). Because of this restriction it is possible to omit the `Lab` annotations on quantified labels. For example, I could have written written the quantifier block for `toString` as `⟨l|α: * @ l|(info α)=l⟩`; from now I will omit them for concision.[1] InforML uses the notation `α: * @ l` for what would be written as $\alpha{:}\star^{\ell}$ in $\lambda_{\text{SEC}i}$.

The last part of the quantifier block, `(info α) = l`, is a label constraint. Because InforML has label polymorphism, and consequently label variables, it is not always possible to directly compare two labels like it is in $\lambda_{\text{SEC}i}$. Initially, the only facts known about a quantified label `l` are that it must be greater than or equal to $\bot$ and less than or equal to $\top$. In many cases, these two facts are not specific enough to show that some code is well-typed. The constraint `(info α) = l` means that to instantiate the `toString` function, it must be that case that the label `info α` is equal to `l`. The label `info α` has a similar meaning to the type meta-operator $\mathcal{L}(\cdot)$ in $\lambda_{\text{SEC}i}$. I will explain the label `info α` in more detail in the coming subsection.

Finally, following the quantifier block is the type of the function itself, `α -(l|⊥)-> String @ l`. In $\lambda_{\text{SEC}i}$, toString was a function with the type $(\alpha)\ @\ \ell \xrightarrow{\bot} (\textbf{string})\ @\ \ell$. There are two differences between the types: there is no label on `α` in InforML, and the function types in InforML have two labels instead of just one. The fact that `α` is unlabeled is related to `info` labels. For now, think of values of type `α` as having an unspecified information content.

1. Why would I ever write the `Lab` annotation? I conjecture that it makes the types slightly more readable to someone completely unfamiliar with InforML. Additionally, when writing label functions, `λ l:Lab =(π)=> τ end`, and their kinds, `Lab -(π)-> κ`, writing `Lab` is required, so there is symmetry in allowing it to be written in quantifier blocks.

The second difference, that there are two labels on the function type in InforML, `l` and $\bot$, versus just the single label $\bot$ in λ_{SECi}, is a consequence of InforML being an impure language. The first label in the InforML version, `l`, specifies the *program counter* of the function. The program counter label is a precondition on the contexts in which the function may be executed; I will give a detailed explanation of the program counter in the coming subsections. The second label in the InforML version, $\bot$, is the information content associated with the function's closure, and has the same meaning as the label on function types in λ_{SECi}. If this second label is $\bot$, InforML allows the label to be omitted. For example, the function type in `toString` could have been written as `α -(l)-> String @ l`.

While the domains of the two functions differ, the range of the function types, `String @ l` and (**string**) @ ℓ, appear to be the same aside from their typefaces, but this is misleading. For now it is okay to consider `String @ l` to be equivalent to (**string**) @ ℓ; I will explain how they differ precisely in the coming subsection.

On the line after the type signature for `toString` is its definition. This part of the function is very similar to what would be written in Standard ML, except the part in angle brackets, `⟨l|α⟩`, immediately after the function's name. In InforML, the contents of these angle brackets are used to give names to the label and type variables that the function quantifies over. Therefore, inside the function's arguments and body, the label that the function quantifies over is named `l` and the type it quantifies over is named `α`. The names can be α-varied from the ones used in the type signature. It is not necessary (or allowed) to give the kinds of the type variables, because they will be inferred by local type inference from the type signature. Similarly, the type of `toString`'s argument, `arg`, will be inferred by local inference.

Moving into the body of `toString`, the `typecase` operator in InforML is similar to the one found in λ_{SECi}, but with several practical differences. First, annotating a `typecase` expression with its type is optional. This is, again, because of the use of local type inference. Like λ_{SECi}, the scrutinee of `typecase` must be of base kind.

The most significant deviation from λ_{SECi} in the body of `toString` is that the branches of `typecase` are not fixed in InforML. One or more branches are specified using a language of type patterns, φ, as described in Figure 3·1. Type patterns are similar to the patterns found in other ML-like languages. For example, underscore (`_`) is used as the wildcard pattern. However, InforML does not require type patterns to be linear. That is, patterns can reference already bound variables, and the variables that patterns do bind can be referenced more than once.

The languages of type patterns is more restrictive than the language of types. For example, type patterns do not include quantifiers or type functions. The fact that type patterns cannot contain quantifiers does not diminish the expressive power of InforML because the language is predicative. Therefore, a type variable can never be bound to a quantified type at runtime. Additionally, patterns for type functions would require the use of higher-order matching. Higher-order matching is known to be decidable (Stirling 2006), but it would make the process of compiling InforML and its accompanying runtime system significantly more complicated.

Labels in type patterns are restricted to wildcards and atomic labels (see ℓ versus Л^2 in Figure 3·1) to ensure that pattern matching is tractable and deterministic at runtime. For example, if the type

2. I ran out of meta-variables for label-like entities, so full labels are indicated using the Cyrillic capital letter "el" (Л).

`Int @ l` were matched with the type pattern `Int @ (l1 ⊔ l2)`, where `l1` and `l2` are binding variables, the InforML runtime could not determine a unique decomposition of `l` into a join of two labels.

InforML does not check that pattern matching is exhaustive or whether patterns are overlapping. A failure to match results in a fatal runtime error. This behavior is similar to other ML-like languages, except that a matching failure is usually a recoverable exception rather than a fatal one.

Finally, the code in the branches of `toString` is nearly identical to the $\lambda_{\text{SEC}i}$ version. The only significant difference is that in InforML it is possible to pull the abstraction for `toString`'s value argument outside of the `typecase`. This is because InforML's `typecase` primitive will refine the program context within a branch by introducing new equalities, something that $\lambda_{\text{SEC}i}$ does not do. The remaining differences are mostly syntactic. For example, in InforML the projections **fst** and **snd** are written as `.0` and `.1`, respectively, and instantiating polymorphic functions is written with angle brackets, `⟨ … ⟩`, instead of square brackets, […].

Having finished the overview of the differences between toString, as written in $\lambda_{\text{SEC}i}$, and `toString`, as written in InforML, I will now explain how type constructors and types are merged in InforML and the nature of the program counter in more detail. I will then conclude this section with a detailed explanation of how and why `toString` type-checks in InforML.

§ Combining type constructors and types

As I have mentioned previously, InforML does not differentiate between type constructors and types. Also, in my overview of `toString`, I stated while `String @ l` and (**string**) **@** ℓ appear very similar syntactically, and for practical purposes have the same meaning, these types differ. The difference is in the kinds.

In $\lambda_{\text{SEC}i}$, the type constructor **string** has kind $\star^{\bot}$ while in InforML, the primitive type `String`, has kind `Lab -(+)-> (* @ ⊥)`. For now ignore the + annotation on the arrow kind. In $\lambda_{\text{SEC}i}$, the grammatical distinction between type constructors and types enforces that all types are properly labeled. However, there is no such distinction in InforML, so another mechanism is needed. The mechanism I decided on was to overload the meaning of label application in monotypes. Instead of reading the information content off an injection, like in $\lambda_{\text{SEC}i}$, the convention used in InforML is that the last label application in a monotype is the information content of value with that type. Therefore, `String @ ⊥` and (**string**) **@** $\bot$ mean the same thing, "a string value with an information content of bottom", but are very different in their construction.

The kind of `String` reveals another difference between InforML and $\lambda_{\text{SEC}i}$: variances. In $\lambda_{\text{SEC}i}$, function kinds are labeled with their information content, just like function types. However, because there is only extensional equality on types in InforML, the label has been eliminated and pushed into the type function's range.[3] Instead, in InforML function kinds are annotated with their variance. The variance for `String` is written as `-(+)->`, which means that subtyping should treat the label application in `String @ l` covariantly. That is, `String @ ⊥` is a subtype of the type `String @ ⊤`. A contravariant type or label argument is specified by writing `-(-)->` and invariant arguments are specified by writing `-(±)->`. The reason InforML includes explicit variance annotations is because, unlike $\lambda_{\text{SEC}i}$, the language of types

3. The label on kind functions in $\lambda_{\text{SEC}i}$ probably should have been dropped; while there are languages with pointer equality on values, I am not aware of any language with pointer equality on types. However, Dreyer's (2005) calculus for recursive modules, does allow for a limited form of imperative update on type definitions, so maybe the idea is not too far-fetched.

is not fixed. In λ_{SECi}, it was acceptable to hardwire the variances of each type into the subtyping rules. Programmers can use the variance annotations provided by InforML to specify how subtyping works for their data types.

Another consequence of using the final label application in a monotype to describe the information content of a value is that it if there is no final label application, it is not possible to directly refer to that value's information content. Again, this does not arise in λ_{SECi} because it uses its grammar to enforce that all types are labeled, while InforML does not. As a specific example, the argument type of `toString` is α. When α was quantified over it was specified to have kind `* @ l` and is therefore a well-kinded type describing a value. However, because the type α has no label application the information content of a value with type α cannot be directly named.

InforML resolves this problem by introducing a new form of label, the `info` label, that allows for indirectly referring to a value's information content. For example, where the information content of the argument of toString was directly specified by writing (α) @ ℓ, indicating that it has an information content of ℓ, in InforML its information content is referred to indirectly using the label `info α`. An `info` label can be seen as making the type meta-operator $\mathcal{L}(\cdot)$ from λ_{SECi} part of the language. Like $\mathcal{L}(\cdot)$, the `info` label is only defined for types with base kind.

There are several equivalences that InforML uses to simplify `info` labels. Here are some of these equivalences:

$$\begin{aligned} \texttt{info}\ (\tau\ \texttt{@}\ \Pi) &= \Pi \\ \texttt{info}\ (\tau_1\texttt{-(}\Pi_1\texttt{|}\Pi_2\texttt{)->}\ \tau_2) &= \Pi_2 \\ \texttt{info}\ (\tau_1\ \ldots,\ \tau_n) &= \texttt{info}\ \tau_1 = \cdots = \texttt{info}\ \tau_n \end{aligned}$$

The first equivalence codifies the convention that the information content of a value is equivalent to the last label application in the type. The second equivalence specifies that the information content of a function value is the second label on the function type's arrow. The third equivalence specifies that the information content of a tuple is obtained the same as any one of its components. That means that in order to construct a tuple, subsumption must be used to make the information content of each of its components the same.

The use of `info` labels can be partly avoided by making use of InforML's higher-order type polymorphism. In `toString` the problem arose from the fact that it is not possible to directly refer to the information content of a value with type α, because the type contains no label applications. An alternative strategy when quantifying over α is to give it kind `Lab -(+)-> (* @ l)` instead of kind `* @ l`:

```
fun toString : ∀⟨l:Lab|α: Lab -(+)-> * @ l⟩ α @ l -(l)-> String @ l
fun toString ⟨l|α⟩ arg =
    typecase (α @ l)
           | Bool @ l1      =>
             if arg then "True" else "False" end
           | _ -(_ | _)-> _ =>
             "<Function>"
           | (β @ _, ψ @ _)         =>
             "(" ^ (toString ⟨l|β⟩ (arg.0)) ^ "," ^ (toString ⟨l|ψ⟩ (arg.1)) ^ ")"
```

In this version of `toString` because the information content of the argument is directly specified as `l`, it is possible to eliminate the need for the associated constraint. However, this version of `toString` also has a subtle problem.

In this implementation of `toString`, for the branch for tuple types necessary to use type variables applied to unspecified labels:

```
| (β @ _, γ @ _)          =>
```

This is necessary so that `β` and `γ` will have the kind, `Lab -(+)-> * @ l`, needed to call this version of `toString` recursively.[4] The problem is that the type pattern will only match against types whose normal form ends in a label application. However, function types and tuple types in normal form do not end in a label application. Therefore, if `toString` is invoked as

```
toString ⟨⊥|(λ l:Lab =(+)=> (Int @ l, (Int @ l, Int @ l)) end)⟩ (1, (2, 3))
```

It will abort with a type matching failure because the type `(Int @ ⊥, (Int @ ⊥, Int @ ⊥))` will not match against the type pattern `(β @ _, γ @ _)` (or any of the other type patterns). The other problem with this version of `toString`, though it is more of an annoyance, is that it frequently requires η-expanding the type argument to be a function from labels to types.

Additionally, working with type functions that abstract over labels is complicated by the requirement that the information content of their body be equal to the abstracted label. That is, for the label function `λ l:Lab =(π)=> τ end` to be well-formed, the constraint `(info τ) = l` must be true. This condition is necessary to ensure that type equivalence cannot introduce contradictory label equivalences. A specific example of what can go wrong is the following chain of equivalences

```
Л = info ((λ l:Lab =(+)=> Int @ ⊥ end) @ Л) = info (Int @ ⊥) = ⊥
```

Furthermore, making the relation directed or requiring that `τ` be in weak-head normal form will not fix the problem. Either solution would allow equivalences like `Л = α @ Л`, where `α` is a type variable, because `α @ Л` is already in normal form.

These examples illustrate that the choice between using `info` labels and higher-order type polymorphism is more than just pushing labels around in different ways. Why use `info` labels at all then? In § 5·2, I will discuss how I would have designed InforML knowing what I have learned from its development.

The `info` label was partly inspired by the `level` constraint found in the FlowCaml language (Pottier and Simonet 2003). In § 3·6, I will compare InforML and FlowCaml in detail.

§ The program counter

In my overview of the `toString` function, I noted that in the function type `α -(l|⊥)-> String @ l`, the label `l` annotating the function arrow is its program counter. The program counter acts as precondition on the contexts in which a function may be invoked.

When moving from a *pure* language with an information-flow type system, like λ_{SECi}, to an *impure* language with an information-flow type system it is necessary to extend the type system with the notion

4. I have glossed over the fact that a kind annotation would be truly necessary for `β` and `γ` to also have the correct variance.

of a *program counter*. The most common language extensions that make a language impure are mutable state, non-local control operations, such as continuations and exceptions, non-termination,[5] and I/O. These features introduce what are known as *implicit flows* into an information-flow type system. An implicit flow occurs when a value can depend upon the information content of another value (or in the case of InforML, a type), even though the first value is not directly computed from the second. The following is a typical example, written in Standard ML, of an implicit flow created by a combination of control-flow and mutable state:

```
val x = ref 0              (* x is low security *)
val y = true               (* y is high security *)
fun f () = (x := 1)
fun g () = (x := 2)

if y then f () else g ()   (* x now depends upon y *)
```

To deal with this problem, the type system assigns each control-flow point in the program a label representing the information that has been learned as a consequence of execution reaching that point in the program. Thus the name "program counter label", which evokes the idea of the memory address that the CPU is currently executing.

The program counter label is used in two ways by the type system. Firstly, all manipulated values must have an information content at least as high as the current program counter. For example, if the current program counter is `l1` then an integer `42` in InforML must be given a type `Int @ l2` where `l2` is greater than or equal to `l1`. The reason for this requirement is that it accounts for the fact that the current trace of a program's control flow has as significant of an impact on the value as the fact that it may have been computed by multiplying `(6 : Int @ l1)` by `(7 : Int @ ⊥)`.

Secondly, control-flow transfers to points within a program that have a program counter lower than the current program counter label are disallowed. Therefore, if the current program counter is Π_1 then it is only possible to invoke a function with the type τ_1 `-(`Π_2`|`Π_3`)->` τ_2 if Π_2 is greater than or equal to both Π_1 and Π_3. However, when control is transferred to a location with a higher program counter, such as through a function call or conditional, once execution returns to the point where the transfer was initiated, it is allowable to restore the program counter to its previous state. This is safe because all control flow paths within a function or conditional must return to the same context from which the function was called or the conditional executed. Unrestricted continuations (Sitaram and Felleisen 1990) and control-flow operators like "goto" (Dijkstra 1968) do not always have this property.

The following code fragment illustrates how the program counter label changes with the control-flow of a conditional:

5. To date, most realistic languages with information-flow type systems do not consider non-termination as an effect, even though it is a potential source of implicit flows. As this is an orthogonal research problem, InforML also ignores implicit flows caused by abnormal termination and non-termination.

```
# Program counter is l1
...
if (mybool : Bool @ l2) then
  # Program counter is now (l1 ⊔ l2)
  ...
else
  # Program counter is now (l1 ⊔ l2)
  ...
end
# Program counter is now l1 again
...
```

For simple programs, there is little difference between joining the result type of a conditional with the information content of the scrutinee, like in $\lambda_{\text{SEC}i}$, and the use of a program counter. However, in larger programs it makes a significant difference. This difference is illustrated in the following example:

```
# fun foo : Int @ ⊤ -> Int @ ⊥
val (bar : Int @ ⊤) = if (h : Bool @ ⊤) then
                        foo 1
                      else
                        0
```

I have omitted the program counter label on the function arrow of `foo` because it is not directly relevant to the example, and could prove confusing. If InforML typechecked conditionals like $\lambda_{\text{SEC}i}$, it would determine that each branch in the above conditional has type `Int @ ⊥` It would then join that type with the information content of the scrutinee, `⊤`, to give `bar` the type `Int @ ⊤`, just like in the annotation above. However, because InforML uses a program counter, while typechecking the branches the result of the call to `foo` will be `Int @ ⊥` but it will be immediately raised to `Int @ ⊤` because it must have an information content greater than or equal to the program counter label. Similarly, when `0` is type-checked it will be given type `Int @ ⊤` to ensure that its information content is greater than the program counter label. So in the end, `bar` still receives the type `Int @ ⊤`.

However, changing the domain of `foo` will have a significant impact:

```
# fun foo : Int @ ⊥ -> Int @ ⊥
...
```

The declaration for `bar` will continue to type-check in $\lambda_{\text{SEC}i}$, where the result type of the conditional is joined with the information content of the scrutinee, but fail in InforML where a program counter label is used. The reason that `bar` fails to typecheck in InforML is because while typechecking `1` it will determine that it has the type `Int @ ⊤`. Because `Int @ ⊤` is not a valid argument for `foo` with its revised type, the sub-expression `foo 1` now fails to typecheck.

The way conditionals raise the program counter, as described above, also applies when typechecking the `case` and `typecase` primitives in InforML.

§ `toString` in detail

Now that I have finished reviewing the differences between λ_{SEC_i} and the core of InforML in detail, I can return to the implementation of `toString` and explain in detail how and why it typechecks. Recall the definition of `toString`:

```
fun toString : ∀⟨l:Lab|α: * @ l|(info α) = l⟩ α -(l|⊥)-> String @ l
fun toString ⟨l|α⟩ arg =
    typecase α
           | Bool @ l      =>
             if arg then "True" else "False" end
           | _ -(_ | _)-> _ =>
             "<Function>"
           | (β, ψ)         =>
             "(" ^ (toString ⟨l|β⟩ (arg.0)) ^ "," ^ (toString ⟨l|ψ⟩ (arg.1)) ^ ")"
         end
```

The first branch of `toString` will match when the scrutinee is the boolean type applied to the label `l`.

```
...
           | Bool @ l   =>
             if arg then "True" else "False" end
...
```

If this branch matches, it will be the case that `α` is equal to the type `Bool @ l`, and therefore it is possible to perform a conditional dispatch on `arg`. The conditional will raise the program counter label from `l`, as specified by `toString`'s type signature, to `(l ⊔ l)` by `l`, because `info (Bool @ l) = l`. However, `(l ⊔ l)` is just equivalent to `l`. Therefore, the strings that are returned have the required type `String @ l`.

It is okay that I used type pattern `Bool @ l` rather than `Bool @ _` or `Bool @ l'` for some fresh label variable `l'`, because of the precondition on `toString`. Given that `(info α) = l`, for any label pattern `lp`, matching with the pattern of `Bool @ lp` will imply that `α = Bool @ lp` which means that `(info (Bool @ lp) = lp = l`. So, by definition `Bool` can only be applied to the label `l`.

The second branch of `toString` is straightforward.

```
...
           | _ -(_ | _)-> _ =>
             "<Function>"
...
```

InforML has no mechanism for intensionally analyzing functional values, so it just returns the string `"<Function>"`.

The final branch of `toString` will match when the scrutinee is a two element tuple type.

```
...
           | (β, γ)      =>
             "(" ^ (toString ⟨l|β⟩ (arg.0)) ^ "," ^ (toString ⟨l|γ⟩ (arg.1)) ^ ")"
...
```

monotypes	τ	::=	…	
		|	m.α	module type projection
expressions	e	::=	…	
		|	m.x	module value projection
declarations	d	::=	ld	local declaration
		|	module m (: S)? = M	module declaration
		|	signature s = S	signature declaration
		|	type α_1 (: κ_1)? = τ_1 and … and α_n (: κ_n)? = τ_n	type definitions
modules	M	::=	m	module variable
		|	mod d* end	
signature bindings	sb	::=	type α : κ	opaque type definition
		|	type α : κ = τ	translucent type definition
		|	val x : σ	values
		|	fun x : σ	functions
signatures	S	::=	s	signature variables
		|	sig sb* end	

Figure 3·2: The grammar of the InforML module language.

Here, typechecking the recursive calls of `toString` is the most interesting part. By definition, if the type pattern `(β, γ)` has kind `* @ l` then the patterns β and γ have kind `* @ Л`, for some Л, where `Л <: l`. To call `toString` recursively on values with type β and γ, the constraints `(info β) = l` and `(info γ) = l` must hold. However, given that executing this branch implies that `α = (β, γ)`, and the precondition `(info α) = l`, by the definition of `info` described previously, it is the case that `(info (β, γ)) = (info β) = (info γ) = l`.

Now that I have finished explaining how the core of InforML differs from $\lambda_{\text{SEC}i}$, I will turn to covering features of InforML that have no analog in $\lambda_{\text{SEC}i}$.

§ 3·2 Modules

In $\lambda_{\text{SEC}i}$ abstract data types can be simulated by using open terms with free type and term variables. InforML provides a more practical solution in the form of a simple module system.[6] Figure 3·2 shows the portion of the InforML grammar for modules that was elided from Figure 3·1.

To illustrate the use of modules in InforML, I will examine the implementation of a module for rational numbers. An initial implementation might look like the following:

6. Modules in InforML, like most members of the ML family of languages, are second-class dependent records. InforML, however, does not provide second-class functions over modules, often called *functors*. Functors would be a useful addition to InforML, but I believe there is nothing interesting from a research standpoint in adding them to InforML.

```
module rational = mod
  fun fromInt : Int @ ⊥ -(⊥)-> (Int @ ⊥, Int @ ⊥)
  fun fromInt i = (i, 1)

  fun toInt : (Int @ ⊥, Int @ ⊥) -(⊥)-> Int @ ⊥
  fun toInt (n, d) = n div d

  fun mult : (| (Int @ ⊥, Int @ ⊥), (Int @ ⊥, Int @ ⊥) |) -(⊥)-> (Int @ ⊥, Int @ ⊥)
  fun mult (n1, d1) (n2, d2) = (n1 * n2, d1 * d2)
end
```

This module implements rational numbers as a pair of integers, with the numerator as the first component and the denominator as the second component. The function `fromInt` takes an integer and converts it to a rational number by giving it a denominator of `1`. The function `toInt` gives an approximation of a rational number as an integer by dividing the numerator by the denominator. Finally, `mult` provides a means of multiplying rational numbers.

With this example I use a shorthand notation for curried function types. In InforML, "banana braces" are used as syntactic sugar for curried function types. Any function type written as

$$(|\tau_1, \ldots, \tau_n|)\ \text{-}(\Pi_1|\Pi_2)\text{->}\ \tau$$

expands during typechecking to the longer type

$$\tau_1\ \text{-}(\Pi_1|\Pi_2)\text{->} \ldots \text{-}(\Pi_1|\Pi_2)\text{->}\ \tau_n\ \text{-}(\Pi_1|\Pi_2)\text{->}\ \tau$$

When the module `rational` is typechecked, InforML will infer the following signature:

```
sig
  fun fromInt : Int @ ⊥ -(⊥)-> (Int @ ⊥, Int @ ⊥)
  fun toInt : (Int @ ⊥, Int @ ⊥) -(⊥)-> Int @ ⊥
  fun mult : (Int @ ⊥, Int @ ⊥) -(⊥)-> (Int @ ⊥, Int @ ⊥) -(⊥)-> (Int @ ⊥, Int @ ⊥)
end
```

This signature is uninteresting because it is just the collection of the type signatures I have written for the functions, modulo canonicalization. Furthermore, the signature completely exposes the implementation of rational numbers. There is nothing preventing a user from creating a pair of integers that does not correspond to a valid rational number, for example, a rational number with a denominator of `0`.

These deficiencies can be resolved in two steps. First, the `rational` module needs to define what is meant by a rational number. This can be done by adding a type definition.

```
module rational = mod
  type t = λ l:Lab =(+)=> (Int @ l, Int @ l) end

  fun fromInt : Int @ ⊥ -(⊥)-> t @ ⊥
  fun fromInt i = (i, 1)

  fun toInt : t @ ⊥ -(⊥)-> Int @ ⊥
  fun toInt (n, d) = n div d

  fun mult : (| t @ ⊥, t @ ⊥ |) -(⊥)-> t @ ⊥
  fun mult (n1, d1) (n2, d2) = (n1 * n2, d1 * d2)
end
```

This revised version of the `rational` module gives a definition for a type variable `t`. It is described as a covariant type function from a label `l` to pair of integers with an information content of `l`. Analogous to function kinds, the arrow written `=(+)=>` between the type function arguments and the type function body is annotated with the variance of the type function. If the type function is intended to be covariant, like `t`, the argument can only appear in positions within the body where it can vary covariantly with respect to subtyping and subkinding. For example, in the type

```
λ l:Lab =(+)=> (Int @ l, Int @ l) end,
```

the label `l` occurs covariantly, but in

```
λ l:Lab =(+)=> (Int @ l -(⊥)-> Int @ ⊥) end,
```

the label `l` occurs contravariantly because it appears in the domain of a function type. If a variable is used both co- and contravariantly, I say that it occurs invariantly.[7]

In this revised version of the `rational` module, I have also changed the specifications for `fromInt`, `toInt`, and `mult` by replacing each occurrence of `(Int @ ⊥, Int @ ⊥)` with `t @ ⊥`. This is allowed because inside the module the variable `t` is known to be equal to the type

```
λ l:Lab =(+)=> (Int @ l, Int @ l) end
```

and, by equivalence, the type

```
(λ l:Lab =(+)=> (Int @ l, Int @ l) end) @ ⊥
```

β-reduces to the type `(Int @ ⊥, Int @ ⊥)`.

InforML will now infer the following signature for my revised version of the `rational` module:

```
sig
  type t : Lab -(+)-> (* @ ⊥) =
         λ l:Lab =(+)=> (Int @ l, Int @ l) end
  fun fromInt : Int @ ⊥ -(⊥)-> t @ ⊥
  fun toInt : t @ ⊥ -(⊥)-> Int @ ⊥
  fun mult : t @ ⊥ -(⊥)-> t @ ⊥ -(⊥)-> t @ ⊥
```

7. Formally, for a variable to appear covariantly means that the variable occurs positively, while a variable that appears contravariantly occurs negatively.

This signature says that `t` is a type variable with kind `Lab -(+)-> (* @ ⊥)` and that it is equal to the type `(λ l:Lab =(+)=> (Int @ l, Int @ l) end)`.

While the `rational` module now has a defined notion of what it means for a value to be a rational number, it still does not provide any data abstraction. The next step is to ascribe `rational` with a signature that does not expose the implementation of rational numbers to the rest of the program. Extending `rational` with such a signature looks like the following:

```
module rational : sig
  type t : Lab -(+)-> (* @ ⊥)

  fun fromInt : Int @ ⊥ -(⊥)-> t @ ⊥
  fun toInt : t @ ⊥ -(⊥)-> Int @ ⊥
  fun mult : (| t @ ⊥, t @ ⊥ |) -(⊥)-> t @ ⊥
end = mod
  ...
end
```

In this signature, I have changed `t` from a *translucent* type signature to an *opaque* type signature (Harper and Lillibridge 1994). A type signature is called translucent when it reveals its definition. A type signature is opaque when it does not reveal its definition. To make a type definition opaque, all that must be done is to leave off the `= ...` part of the signature that follows the kind.

With the above signature signature ascription it is not directly possible to provide the `rational` module with invalid instances of a rational number. That is, `rational.toInt (1, 0)` is ill-typed because outside the `rational` module, the type `rational.t @ ⊥` is not equal to, or even a supertype of, the type `(Int @ ⊥, Int @ ⊥)`.

However, the `rational` module is still vulnerable to having its integrity violated using `typecase`. While calling `rational.toInt` on `(1, 0)` directly is now ill-typed, it is still possible to cause a divide-by-zero exception by writing the following bit of code:

```
typecase rational.t @ ⊥
       | (Int @ ⊥, Int @ ⊥) =>
           rational.toInt (1, 0)
```

Inside the `typecase` branch it is known that the type `rational.t @ ⊥` is equivalent to the type `(Int @ ⊥, Int @ ⊥)`. Therefore, `rational.toInt (1, 0)` will be well-typed, and when executed `rational.toInt` will attempt to convert `(1, 0)` to an integer by dividing `1` by `0`. However, this will cause the program aborting with a divide by zero error.

A more restrictive signature for the `rational` module can prevent the above abuse. For example, I could have given the abstract type `rational.t` a more restrictive kind:

```
module rational : sig
  type t : Lab -(+)-> (* @ ⊤)
  ...
end = mod
  ...
end
```

With this new signature for `rational`, the programmer can know that if a program expression `e` has type `τ`, and `info τ` is equivalent to ⊥ or any label other than ⊤, then if `e` evaluates to a value, that value does not depend upon the definition of `rational.t`. That is, `e` is parametric in the definition of `rational.t`.

Furthermore, this change prevents my example expression from violating `rational`'s integrity. Because `rational.t @ ⊥` now has kind `* @ ⊤`, inside the branches of `typecase` the program counter is raised to ⊤ when evaluating the original expression:

```
typecase rational.t @ ⊥
       | (Int @ ⊥, Int @ ⊥) =>
           # rational.t @ ⊥ = (Int @ ⊥, Int @ ⊥) and program counter label is ⊤
           rational.toInt (1, 0)
```

Because the program counter label is ⊤, the value `(1, 0)` now has type `(Int @ ⊤, Int @ ⊤)`. However, `rational.toInt` still has the type `rational.t @ ⊥ -(⊥)-> Int @ ⊥`, where `rational.t @ ⊥` is known to be equivalent to `(Int @ ⊥, Int @ ⊥)`. Therefore, the function application is no longer well-typed.

In this example, the fact that changing the label on the kind of `rational.t` prevented the integrity of `rational`'s abstraction from being violated is somewhat accidental. A more realistic implementation of rational numbers would have made `toInt` label polymorphic,

```
  ...
  fun toInt : ∀⟨l⟩ t @ l -(l)-> Int @ l
  fun toInt ⟨l⟩ (n, d) = n div d
  ...
end
```

This implementation is more realistic because quantifying over the information content of the input, program counter, and the output, allows `toInt` to be used in all program contexts, rather than only those where the program counter is ⊥ and a rational with an information content of ⊥ is available. However, this change makes it possible to rewrite the original expression so that integrity can be violated:

```
typecase rational.t @ ⊤
       | (Int @ ⊥, Int @ ⊥) =>
           rational.toInt ⟨⊤⟩ (1, 0)
     end
```

(Here, I have made the label instantiation of `rational.toInt` explicit to make the example clearer; the instantiation can be inferred using local inference) It is still possible to recover integrity using techniques that I will describe in § 4.

In practice it is desirable for benign uses of type-directed programming in InforML to be able to distinguish between rational numbers and data that just happens to be a pair of integers. The standard solution to this problem is to use generative data types. In the next section, I will explain how generative data types work and interact with type-directed programming in InforML.

§ 3·3 Generative data types

In § 1·4, one solution to the problems presented by reflection that I examine is the use of type generativity. While I deemed type generativity to be an unsuitable foundation for reasoning about both confidentiality

kinds	κ	::=	…	
		|	`% @ Л`	algebraic type classifiers
monotypes	τ	::=	…	
		|	`A`	algebraic data types
type patterns	φ	::=	…	
		|	`A`	algebraic data types
term patterns	p	::=	…	
		|	$\texttt{D}\,(\langle \texttt{l}^* \mid \alpha^* \rangle)^?\ \texttt{p}_1 \ldots \texttt{p}_n$	data constructors
expressions	e	::=	…	
		|	$\texttt{D}\,(\langle \Л^* \mid \tau^* \rangle)^?$	instantiation
		|	$\texttt{isdata}\ \alpha\ \texttt{then}\ \texttt{e}_1\ \texttt{else}\ \texttt{e}_2\ \texttt{end}$	downcast
datatype bindings	dtb	::=	$\texttt{A : } \kappa = (\mid)^?\ \texttt{D}_1 : \sigma_1 \mid \ldots \mid \texttt{D}_n : \sigma_n$	
declarations	d	::=	…	
		|	$\texttt{datatype}\ \texttt{dtb}_1\ \texttt{and}\ \ldots\ \texttt{and}\ \texttt{dtb}_n$	data type definitions
signature bindings	sb	::=	…	
		|	`data A : κ`	algebraic data type
		|	`con D : σ`	constructor binding

Figure 3·3: The grammar for InforML's generative data types.

and integrity independently, it is still useful in practice. For example, even though rational numbers may be implemented as a pair of integers, like in the previous section, when writing type-directed operations it may be important to the semantics of the operation that rational numbers and pairs of integers be treated differently. Like most ML-like languages, InforML provides generative algebraic data types – each new algebraic data type is not equivalent to any other type. Figure 3·3 shows the parts of the InforML language relating to generative data types that were elided from Figure 3·1.

As a simple example of an algebraic data type, I will start with the definition of a binary tree structure that contains no data:

```
datatype Tree : Lab -(+)-> (% @ ⊥) =
   | Leaf : ∀⟨l⟩ Tree @ l
   | Node : ∀⟨l⟩ Tree @ l -(⊤|⊥)-> Tree @ l -(⊤|⊥)-> Tree @ l
```

Unlike type definitions, InforML requires a kind annotation when defining a new algebraic data type – it is not always possible for local inference to synthesize the kind of an algebraic data type from its definition. The `Tree` data type has been defined to have the kind `Lab -(+)-> (% @ ⊥)`.

The `Tree` data type can be described as type function from labels to types of *algebraic kind* with information content of `⊥`. InforML makes a distinction between types with *type kind* (`*`) and those with *algebraic kind* (`%`). This distinction is only necessary so that it is possible to restrict some operations to only work on algebraic data types and their associated data constructors. The kind `% @ Л` is a subkind of `* @ Л` for all `Л`.

I have given the `Leaf` data constructor the type `∀⟨l⟩ Tree @ l`. This type means that `Leaf` for any label `Л`, constructs a value of type `Tree @ Л` – a tree whose structure has an information content of `Л`.

The `Node` constructor has the type

```
∀⟨l⟩ Tree @ l -(⊤|⊥)-> Tree @ l -(⊤|⊥)-> Tree @ l,
```

This type can be understood to mean that if the current program counter is less than `⊤`, then for any label `l` when `Node` is applied to two values of type `Tree @ l`, it builds a value of type `Tree @ l`. In InforML, data constructors are not compiled to functions, despite having a functional types. Therefore, the data constructor's program counter labels are always `⊤` and function closure labels are always `⊥`, ensuring that it is always possible to construct a new value from a data constructor. The function closure labels on data constructors are required to be `⊥` to maximize their reuse.

It is tempting to try giving `Leaf` the more concise type `Tree @ ⊥` because the `Tree` data type is covariant and therefore it would be possible to use subsumption to give `Leaf` type `Tree @ Л` for any label `Л`, seemingly the same as is possible with the type `∀⟨l⟩ Tree @ l`. However, there is a subtle difference. A type like `Tree @ ⊥` is called an indexed type or GADT (Coquand 1992; Crary and Weirich 1999; Xi, Chen, and Chen 2003; Peyton Jones, Vytiniotis, Weirich, and Washburn 2006), because the arguments of `Tree` are not parametric. The index in type `Tree @ ⊥` is the label `⊥`. In InforML, a type with label index is always equivalent to a type where the argument is universally quantified, but constrained by an equality. For example, giving `Leaf` the type `Tree @ ⊥` is equivalent to giving it the constrained polymorphic type `∀⟨l|l = ⊥⟩ Tree @ l` not the type `∀⟨l|⊥ <: l⟩ Tree @ l`. In most cases, the programmer probably does not intended that `Leaf` values can only have the type `Tree @ ⊥`. Therefore, I chose to give `Leaf` the type `∀⟨l⟩ Tree @ l` because it is the most general type.

Now that I have explained the basics of algebraic data types in InforML, I will move on to more complex algebraic data types.

§ Dependent kinds

The `Tree` data type is one of the simplest recursive data structures that can be defined in InforML. I chose it primarily to focus on a few key concepts. However, more complex data structures in InforML, parametrized containers for instance, often require *dependent kinds*.

The simplest non-trivial container structure is the "option" data type. In InforML, it is defined as the following:

```
datatype Option : Π l:Lab -(+)-> (* @ l) -(+)-> Lab -(+)-> (% @ l) =
   | None : ∀⟨l1 l2|α: * @ l1⟩ Option @ l1 α @ l2
   | Some : ∀⟨l1 l2|α: * @ l1|(info α) = l2⟩ α -(⊤)-> Option @ l1 α @ l2
```

The type `Option @ l1 α @ l2` can be interpreted as a value possibly containing a value of type `α`, where `α` has kind `* @ l1`, and the information content of the overall value is `l2`.

The kind of `Option`,

```
Π l:Lab -(+)-> (* @ l) -(+)-> Lab -(+)-> (% @ l),
```

means that `Option` is a covariant dependent type function from a label `l`, to a covariant type function with a domain accepting types of kind `* @ l`, to a covariant type function from labels to types with kind `% @ l`. The kind of `Option` is dependent in the sense that when the `Option` algebraic data type is fully applied as `Option @ l1 α @ l2`, the acceptable kinds for this type `α` *depend* upon `l1` and the overall kind of `Option @ l1 α @ l2` *depends* upon `l1`.

Because of this dependency, local type inference in InforML allows the above definition to be written as:

```
datatype Option : Π l:Lab -(+)-> (* @ l) -(+)-> Lab -(+)-> (% @ ⊥) =
   | None : ∀⟨l1 l2|α: * @ l1⟩ Option α @ l2
   | Some : ∀⟨l1 l2|α: * @ l1|(info α) = l2⟩ α -(⊤)-> Option α @ l2
```

In the quantifier block, α is already specified to have kind `* @ l1`. Because the label of `Option`'s second (type) argument depends on its first (label) argument, the InforML typechecker can conclude that the missing label argument should be `l1`.

InforML is already a very expressive language, so it is natural to wonder whether dependent kinds are truly necessary. If InforML did not have dependent kinds, the most general definition for the `Option` data type would be the following:

```
datatype Option' : (* @ ⊤) -(+)-> Lab -(+)-> (% @ ⊥) =
   | None' : ∀⟨l|α: * @ ⊤⟩ Option' α @ l
   | Some' : ∀⟨l|α: * @ ⊤|(info α) = l⟩ α -(⊤)-> Option' α @ l
```

This definition, instead of specifying the kind of the type argument to be dependent on a label argument, requires it to have kind `* @ ⊤`. Every fully applied type, and algebraic data type, can be given kind `* @ ⊤` using subsumption so this definition can still be used as a container for values of any type. However, code that uses this definition of `Option'` is too conservative in tracking information flows to be reusable.

Whenever `Option'` is applied to a type, the precise information content of that type is lost to the type system. For example, the type `Int @ ⊥` has kind `* @ ⊥`, but in the partially applied type `Option (Int @ ⊥)` the inversion principles for the InforML kind system can only derive that `Int @ ⊥` has the kind `* @ Л`, for some Л less than or equal to ⊤.

Therefore, type patterns involving the `Option'` algebraic data type will be similarly conservative. The following snippet illustrates this problem with an extension to my earlier example of `toString`.

```
fun toString : ∀⟨l|α: * @ l|(info α) = l⟩ α -(l)-> String @ l
fun toString ⟨l|α⟩ arg =
    typecase α
           ...
           | Option' β @ l =>
             case arg
                | Some' arg' =>
                  # This branch will be ill-typed
                  "Some' " ^ (toString ⟨⊤|β⟩ arg')
                | None'      => "None'"
              end
         end
```

Within `typecase` branch for `Option'`, the typechecker will need to assume that type variable β has the kind `* @ ⊤`. Recursively calling `toString` on `arg'` will require instantiating the label argument of `toString` with ⊤. But this means that `toString` will return a string with an information content of ⊤. However, `toString` is declared to return a value of type `String @ l`. There is no constraint that `l` be equal to ⊤, and as a result the case will be rejected as ill-typed.

One option would be to change the type of `toString` to return strings with an information content of ⊤, but this would severely restrict its reusability.

Another perspective on the problem is that the kind `(* @ ⊤) -> Lab -> (% @ ⊥)` does not provide a connection between the label in the kind of its type argument (`* @ ⊤`) and its overall kind (`% @ ⊥`). It is not possible to tell from its fully applied kind, `% @ ⊥`, what the information content of its argument happens to be. The algebraic datatype "hides" the information content of its type argument. Algebraic data types that hide information in this fashion are a significant obstacle to making precise static guarantees in InforML.

Using the dependent kind

```
Π l:Lab -(+)-> (* @ l) -(+)-> Lab -(+)-> (% @ l)
```

for the `Option` algebraic data type resolves these issues. It provides a means of referring to the exact information content of its type argument and relating the information content of the argument type and the information content of the algebraic type as a whole. If I revise the `toString` implementation from above to use the actual implementation of the `Option` algebraic data type, it looks like the following:

```
fun toString : ∀⟨l|α: * @ l|(info α) = l⟩ α -(l)-> String @ l
fun toString ⟨l|α⟩ arg =
    typecase α
           ...
           | Option @ l β @ l =>
             case arg
                | Some arg' =>
                  "Some " ^ (toString ⟨l|β⟩ arg')
                | None      => "None"
              end
         end
```

Here, the recursive call on `arg'` will be well-typed because it is known that β has precisely the kind `* @ l`.

While the use of dependent kinds has made it possible for the `toString` function to work with more algebraic data types than would be possible otherwise, in practice it is preferable to write `toString` once and not extend the implementation with new cases every time a new algebraic data type is defined. In the next subsection, I describe the the difficulties that algebraic data types introduce for type-directed programming and then describe the solution used by InforML.

§ Analyzing generative data types

As I discussed in § 1·4, generative types provide a form of static access control for type information. However, this was not the motivation for including them InforML, and in fact works against writing type-directed operations that will apply to all InforML values.

The fact that algebraic data types are generative means that the only way for a type pattern to match against them is to name them explicitly. However, this semantics has the problem that type-directed functions must be revised for every new algebraic data type they need to handle.

I solve this problem in InforML by defining a distinguished algebraic data type for what are called *spines* (Hinze, Löh, and Oliveira 2006; Hinze and Löh 2006). The essence of spines is to provide a stan-

dardized view of arbitrary data constructors. However, writing a function to convert data constructors into this standardized form has the same problem as I described above, that every time a new algebraic data type is defined the function would need to be extended with an additional case. To escape this circularity, InforML provides a primitive function called `toSpine` that will convert any data constructor into its spine form.

The `toSpine` function has the type signature:

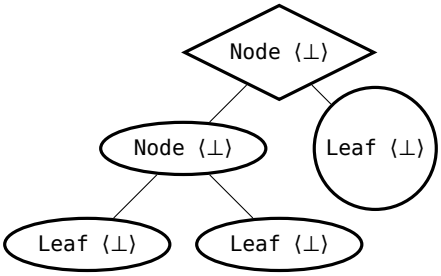
```
fun toSpine : ∀⟨l|α: % @ l|l :> (info α)⟩ α -(⊤)-> Spine α @ l
```

The notable feature of this type is that instead of quantifying over types, it only quantifies over fully applied algebraic data types. The reason for this choice is that it only makes sense to apply `toSpine` to values that are data constructor inhabitants of some algebraic data type.

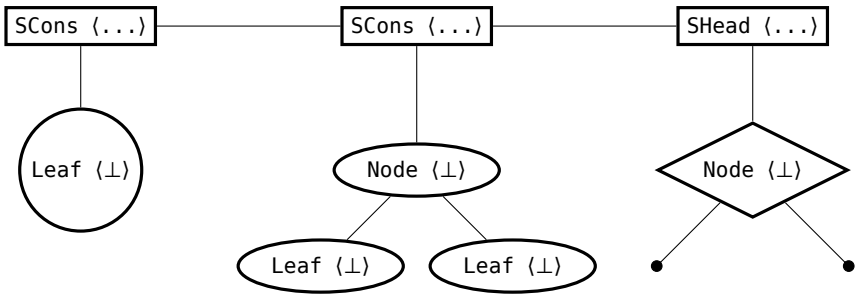
To understand what the `toSpine` primitive does it is helpful to visualize data types diagrammatically. I will use the following value, which has type `Tree @ ⊥`, as an example:

```
Node (Node (Leaf ⟨⊥⟩) (Leaf ⟨⊥⟩)) (Leaf ⟨⊥⟩)
```

I have written `⟨⊥⟩` following `Leaf` to specify the label used to instantiate its quantified label. No such annotation is required for `Node` here as InforML's local type inference algorithm can deduce from its arguments how it should be instantiated. This value of type `Tree @ ⊥` can be visualized as shown below, where the shapes of the nodes have no semantic meaning; they are only intended to make it easier to observe how `toSpine` transforms the structure.



The function `toSpine` converts any data constructor to a value of the `Spine` data type, which has the data constructors `SHead` and `SCons`. For now, I will ignore details of `Spine`'s kind and the types of `SHead` and `SCons`. Calling `toSpine` on the binary tree above yields a `Spine` value that looks like the following:



Notice that `toSpine` has converted the `Tree` to a list-like structure where the root data constructor that was used to build this value, `Node`, is the tail and each argument of `Node` has been added to the

list in reverse order using `SCons`. However, this transformation has only been applied to the head data constructor used construct the value. The child `Node` and `Leaf`s are unchanged. This diagram is equivalent to the InforML value:

```
SCons ⟨...⟩  (SCons ⟨...⟩
                    (SHead ⟨...⟩ (Node ⟨⊥⟩))
                    (Node (Leaf ⟨⊥⟩) (Leaf ⟨⊥⟩)))
             (Leaf ⟨⊥⟩)
```

I have elided the instantiations for `SCons` and `SHead` for the moment. In general, it is not possible for InforML's local type inference algorithm to infer the all the instantiations for a `Spine` data type, even with information provided by expressions containing a `Spine`.

The `SHead` and `SCons` data constructors in InforML are defined using an algebraic data type:

```
datatype Spine : Π l:Lab -(+)-> (* @ l) -(+)-> Lab -(+)-> (% @ l) =
   | SHead : ∀⟨l1 l2|α: * @ l1⟩ α -(⊤)-> Spine α @ l2
   | SCons : ∀⟨l1 l2|(β: * @ l1) (α: * @ l1)|(info β) = l2⟩
                (| Spine (β -(⊤|l2)-> α) @ l2, β |) -(⊤)-> Spine α @ l2
```

The kind of `Spine` is identical to the one I used for `Option` for all the same reasons. The type of the `SCons` data constructor is unusual because the quantified type β, the type of the data constructor argument, does not appear in its result type `Spine α @ l2`. The type variable β can be viewed as existentially quantified in the `Spine` data type.

It is necessary to hide the type of the arguments to a data constructor because there is no guarantee that a fully applied data constructor will have a type that mentions the type of the arguments. For example, consider the following algebraic data type for representing arguments given to a program on a command-line:

```
datatype CmdOpt : Lab -(+)-> (% @ ⊥) =
   | RangeOpt : ∀⟨l⟩ Int @ l -(⊤)-> CmdOpt @ l
   | BoolOpt : ∀⟨l⟩ Bool @ l -(⊤)-> CmdOpt @ l
```

If `toSpine` is used to convert the value `BoolOpt ⟨⊥⟩ True` with the type `CmdOpt @ ⊥` to a spine, it will produce the value `(SCons ⟨...⟩ (SHead ⟨...⟩ BoolOpt) True)` with the overall type of `(Spine (CmdOpt @ ⊥) @ ⊥)`. Because this type is intended to represent `Spine`s for arbitrary instances of `CmdOpt @ ⊥`, there is no way to reveal that the second argument of the `SCons` data constructor type `Bool @ ⊥`.

Returning to my earlier example of showing what `toSpine` would produce when applied to the value `Node (Node (Leaf ⟨⊥⟩) (Leaf ⟨⊥⟩)) (Leaf ⟨⊥⟩)`, I can now fill in the type and label instantiations I had previously omitted:

```
SCons ⟨⊥ ⊥|(Tree @ ⊥) (Tree @ ⊥)⟩
      (SCons ⟨⊥ ⊥|(Tree @ ⊥) (Tree @ ⊥ -(⊤)-> Tree @ ⊥)⟩
             (SHead ⟨⊥ ⊥|(Tree @ ⊥ -(⊤)-> Tree @ ⊥ -(⊤)-> Tree @ ⊥)⟩ (Node ⟨⊥⟩))
             (Node (Leaf ⟨⊥⟩) (Leaf ⟨⊥⟩)))
```

Local type inference allows the value to be expressed more concisely as

```
SCons (SCons ⟨⊥ ⊥|(Tree @ ⊥) (Tree @ ⊥ -(⊤)-> Tree @ ⊥)⟩
             (SHead (Node ⟨⊥⟩))
             (Node (Leaf ⟨⊥⟩) (Leaf ⟨⊥⟩)))
```

The remaining instantiation annotation on the `SCons` data constructor is necessary because it is not possible to synthesize a type for `SHead (Node ⟨⊥⟩)`. Alternately, I could have provided an instantiation annotation for `SHead` instead of `SCons`, but that annotation is slightly longer.

The `Spine` algebraic data type has a third data constructor form that I have not discussed so far. This data constructor, `SConsEx` is necessary because it is not always possible for `toSpine` to construct a `Spine` value from solely `SHead` and `SCons`.[8]

```
datatype Spine : Π l:Lab -(+)-> (* @ l) -(+)-> Lab -(+)-> (% @ l) =
   ...
  | SConsEx :  ∀⟨l1 l2 l3|(β: * @ l2) (α: * @ l1)⟩
                  (| Spine (β -(⊤|l3)-> α) @ l3, β |) -(⊤)-> Spine α @ l3
```

The type of the `SConsEx` data constructor is similar to the type of the `SCons` data constructor. The only differences are that the kinds of β and α are allowed to have differing information content, `l2` versus `l1`, and that `SConsEx` does not constrain the information content of β. Another important point to note is that the quantified label `l2` does not appear in the result type of `SConsEx`, which means that like the type β it is hidden by the data constructor.

The best way to explain why `SConsEx` is needed is through an example, but first, it is necessary to revisit the type of `toSpine`.

```
fun toSpine : ∀⟨l|α: % @ l|l :> (info α)⟩ α -(⊤)-> Spine α @ l
```

The information content of the resulting `Spine`, `l`, is specified as an argument to the function itself. Therefore, the caller can choose the information content of the result, as long as it obeys the constraint that `l :> (info α)`. However, it is possible to call `toSpine` with instantiations for `l` and α that make it impossible for `toSpine` to construct a value of `Spine α @ l` from only the data constructors `SHead` and `SCons`. The example below will illustrate this.

The data constructor `SConsEx` is usually needed to call `toSpine` on existential data types with hidden labels. An example is of this is the *dynamic* type (Abadi, Cardelli, Pierce, and Plotkin 1991) in InforML:

```
datatype Dyn : Lab -(+)-> % @ ⊥ =
  | Dynamic : ∀⟨ld l|α : * @ ld⟩ α -(⊤)-> Dyn @ l
```

Along with `typecase`, the `Dyn` type constructor allows for the emulation of programming idioms found in dynamically typed languages. For example, the `Dyn` type constructor can be used build *heterogeneous lists* of values, that is a list where each element can have a distinct type. However, consider what would happen if the following well-typed InforML code fragment were executed.

```
toSpine ⟨⊥|(Dyn @ ⊥)⟩ (Dynamic ⟨⊤ ⊥|(Int @ ⊤)⟩ 42) : Spine (Dyn @ ⊥) @ ⊥
```

8. The `Spine` data type could be refined further to have four different `SCons` constructors, by creating constructors all of the combinations arising from when the kinds α and β have do and do not have the same information content and when `(info β)` equals the information content of the entire `Spine`. However, I just collapse it down to two data constructors and require that the other cases be distinguished using dynamic constraint checks, which are described in the next section.

The expectation is that `toSpine` would construct the value:

```
SCons ⟨⊥ ⊥|(Int @ ⊤) (Dyn @ ⊥)⟩
      (SHead ⟨⊥ ⊥|(Int @ ⊤ -(⊤)-> Dyn @ ⊥)⟩ (Dynamic ⟨⊤ ⊥|(Int @ ⊤))))
      42
```

I have provided all type and label instantiations for clarity, in practice InforML will be able to infer several of the above instantiations.

Looking back to the definition of `SCons`,

```
...
   | SCons : ∀⟨l1 l2|(β: * @ l1) (α: * @ l1)|(info β) = l2⟩
                (| Spine (β -(⊤|l2)-> α) @ l2, β |) -(⊤)-> Spine α @ l2
...
```

in order for the `SCons` data constructor to construct a value of type `(Spine (Dyn @ ⊥) @ ⊥)`, the label parameter `l2` must be instantiated with ⊥. However, as a precondition, the constraint `(info β) = l2` must be satisfiable. However, to construct a `Spine` for the value `Dynamic ⟨⊤ ⊥|(Int @ ⊤)⟩ 42`, the quantified type argument `β` of `SCons` must be instantiated with the type `Int @ ⊤`. But `info (Int @ ⊤) = ⊤`, so the precondition `(info β) = l2` on `SCons` is unsatisfiable. Therefore, if `toSpine` were to return such a `Spine` value, it would violate type safety.

One solution to this problem would be for `toSpine` to return an `Option Spine`. However, because I prefer that `toSpine` is a total function, the `toSpine` function will use the `SConsEx` data constructor instead of the `SCons` constructor in situations like the one I described above. Therefore, applying `toSpine` to `Dynamic ⟨⊤ ⊥|(Int @ ⊤)⟩ 42` evaluates to the following value:

```
SConsEx ⟨⊤ ⊥ ⊥|(Int @ ⊤) (Dyn @ ⊥)⟩
        (SHead ⟨⊥ ⊥|(Int @ ⊤ -(⊤)-> Dyn @ ⊥)⟩ (Dynamic ⟨⊤ ⊥|(Int @ ⊤))))
        42
```

Because `SConsEx` does not constrain the information content of its second argument to be equal to the overall information content of the value, this value can be safely given the type `Spine (Dyn @ ⊥) @ ⊥`.

It is reasonable to ask, why did I not just give `SCons` the type of `SConsEx`, and eliminate the need for a third data constructor for `Spines`? The reason is that `SConsEx` hides both the information content of the type `β` and the information content of a value with type `β`. Because these information contents are hidden, it is very difficult to reason statically about information-flows when working with values built from `SConsEx`. However, I expect that in typical usage it will be possible to construct most `Spines` using only the `SCons` constructor, which does not hide the information content of any of its components. Consequently, making `SCons` and `SConsEx` distinct allows programmers to make more precise distinctions when working with `Spines`. In § 3·4, I explain InforML's features for supporting dynamic information that can be used ameliorate the problem. In § 3·5, I will give a detailed example of how this works for a realistic example.

§ Downcasting

Finally, it is often necessary to make algebraic data types usable as types by changing their kind from `% @ Л`, for some label `Л`, to `* @ Л`, using subkinding. Consequently, it is no longer possible use `toSpine` on

polytypes	σ	::=	…	
		|	`∃(⟨l:Lab⟩*|⟨α:κ⟩*| C?) σ`	existential quantification
		|	ρ	higher-rank types
higher-rank types	ρ	::=	σ_1 `-(`$\bar{\ell}_1$`|`$\bar{\ell}_2$`)->` σ_2	
term patterns	p	::=	…	
		|	`pack (⟨l*|α*⟩)?` p	existentials
expressions	e	::=	…	
		|	`pack e`	existentials
		|	`ifholds C then` e_1 `else` e_2 `end`	constraint check

Figure 3·4: The grammar for InforML's extensions for dynamic information flow.

values of these algebraic data types. To solve this problem InforML provides a safe *downcasting* primitive called `isdata`:

```
type α : * @ ⊥ = Tree @ ⊥

# The following is ill-typed because α does not have kind % @ ⊥
# val spn = toSpine ⟨⊥ ⊥|α⟩ (Leaf ⟨⊥⟩)

val spn = isdata α then
            # Okay, because α has kind % @ ⊥ in this branch
            toSpine ⟨⊥|α⟩ (Leaf ⟨⊥⟩)
          else
            abort "α is not an algebraic data type."
```

If the scrutinee of `isdata` really is an algebraic data type at runtime, the first branch will be executed, otherwise the second branch will be executed. In the first branch, the typechecker will assume that `α` has kind `% @ ⊥` instead of `* @ ⊥`.

§ 3·4 Dynamic information flow

Even with InforML's highly expressive type system, there are still some cases where it is not possible to express some desirable information-flow policies statically. Consequently, InforML provides two features that make it possible to fall back to tracking information flows dynamically: first-class existential quantification and dynamic constraint checking. The grammar for these features is described in Figure 3·4.

InforML's algebraic data types can be used to express existential quantification, but first-class existential quantification allows programmers to avoid littering their code with new algebraic data type definitions every time they need to existentially quantify. Existential quantification in InforML is written nearly identically to universal quantification. The `∀` preceding a quantifier block is replaced by `∃`, and, unlike universally quantified values, existentially quantified values must be labeled. For example, a function to open a file on disk might be given the following type in InforML:

```
fun openFile : ∀⟨l1⟩ String @ l1 -(l1)-> (∃⟨l2|α: * @ l2|l1 <: (info α)⟩ α) @ l1
```

The range of the `openFile` function is an existentially quantified value. Because InforML program has no knowledge of the structure of the data stored in the file, or its information content, it can only assume

that it has *some* type α of kind `* @ l2` for *some* label `l2` and that α has *some* information content greater than `l1`.[9] Furthermore, it is necessary to label the existential type with `l1` because, as I will explain shortly, it would be possible to introduce illegal flows using existentially quantified labels.

Existentially quantified data is introduced using the `pack` expression. For example, an integer's label can be existentially quantified over by writing `(pack 42) : (∃⟨l⟩ Int @ l) @ ⊥`. The type annotation is necessary here because it is not in general possible to determine which parts of a type are to be hidden. However, because of local type inference in InforML it is not always necessary to directly annotate the `pack` expression. For example, hiding the label on an integer can be generalized to a function:

```
fun hideLabel : ∀⟨l1⟩ Int @ l1 -(⊥|⊥)-> (∃⟨l2⟩ Int @ l2) @ ⊥
fun hideLabel ⟨l1⟩ i = pack i
```

Because the InforML typechecker knows that the body of `hideLabel` must have the type `(∃⟨l2⟩ Int @ l2) @ ⊥`, it is not necessary to annotate the `pack` expression directly.

Existentially quantified data is unpacked using pattern matching. An existentially quantified integer could be incremented as follows:

```
val (newi : (∃⟨l⟩ Int @ l) @ ⊥) =
    case (hideLabel ⟨⊥⟩ 41)
       | pack ⟨l1⟩ i => pack (i + 1)
```

However, because the typechecker has no knowledge of what the label `l1` is within the `case` branch, so that `l1` does not escape its scope it is necessary to package the integer back up in an existential before it can be returned.

Existential quantification allows information-flows to be tracked dynamically. However, as we saw in the example above, once a programmer begins using data or types with existentially quantified labels, often the only way she can avoid repeatedly unpacking and repacking data using these labels is to discard the data. Therefore, InforML provides dynamic constraint checking as a means to recover static checking. For example, the above example could be rewritten as follows:

```
val (newi : Int @ ⊥) =
    case (hideLabel ⟨⊥⟩ 41)
       | pack ⟨l⟩ i =>
           ifholds l <: ⊥ then
             (i + 1 : Int @ ⊥)
           else
             abort "The information content is restricted."
           end
     end
```

Here the `ifholds` primitive allows a programmer to check whether a constraint holds at runtime. While the execution is determined dynamically, the `ifholds` primitive allows the typechecker to make additional assumptions statically. In the example above, the first branch of `ifholds` will only execute if `l = ⊥`, therefore when type checking `i + 1` it can be assumed that the label `l` is equal to ⊥. However, it is necessary to annotate `i + 1` so that the entire expression will be well-typed. Because `ifholds` will not

9. For expository purposes, I am glossing over how types would be preserved when writing data to a file.

try to give `i + 1` a minimal type, `i + 1` by itself will have the type `Int @ l`. However, this is not a valid type for the entire expression because `l` is an existentially quantified variable, and it may not leave the scope of the `case` expression. However, because `l <: ⊥` inside of the `ifholds` expression it is allowed to use type ascription to give `i + 1` the type `Int @ ⊥`. Because InforML does not supported negated constraints, it is not generally possible to assume any relationship between the labels in the `else` branch of an `ifholds` expression.

As I mentioned earlier, unlike universally quantified types, existentially quantified types in InforML must be labeled. Otherwise, when existentially quantified types are combined with dynamic constraint checking it would be possible to subvert the information-flow system. Even though existentially quantified types are very similar to a dependent pairs, it is not enough to just push the information content into the components of existential quantifier like would be done with a tuple.[10]

The following code fragment demonstrates how labels could be used as a covert channel.

```
val (leak: ∃⟨l⟩ Option (Int @ l) @ ⊤) =
    if (h: Bool @ ⊤) then
      pack (None : Option (Int @ ⊥) @ ⊤)
    else
      pack (None : Option (Int @ ⊤) @ ⊤)
    end

val (expose: Bool @ ⊥) =
    case leak
       | pack ⟨l⟩ _ =>
         ifholds l = ⊥ then
           True
         else
           False
         end
```

In this example, if there were no label on the existential package, it could be unpacked, its contents ignored, and label analysis used to decode the value of the boolean `h`. Labeling existential packages ensures that it is not possible to use labels as a covert channel in this fashion.

In the following section, I will conclude this chapter by showing how everything I have described so far can be used to write a version of `toString` that can operate on values of arbitrary type, unlike the implementation I described in § 3·1

§ 3·5 Putting it all together

Now that I have finished reviewing InforML, it is worthwhile to give some example that combines all of the features I have discussed. I will first give a more realistic implementation of `toString` and show how the example in Figure 1·5, found in § 1·3, would be written in InforML.

10. This is only because the first component of the "pair" that an existentially quantified type forms may contain a label, and labels in InforML do not have an information content. Zheng and Myers (Zheng and Myers 2004) resolve this problem by working with labels reified as values that can be given an information content. Therefore, labels in their system can have an information content.

Returning to my running example of "to string", Figure 3·5 shows a module of type-directed functions that provides a version of `toString` that will work for all values in InforML.[11]

Unlike my prior versions of `toString`, this implementation is defined mutually-recursively with a helper function called `spineToString`. For algebraic data types, `toString` converts them to `Spine`s and hands them off to `spineToString`. Because `spineToString` is intended to be used only by `toString`, its type signature is hidden by `tdp`'s module signature.

In the definition of `toString` the cases for booleans, functions, and pairs are essentially the same as in my original implementation. The only difference is that I have replaced all the unnecessary label and type variable binders with wildcards.

The `toString` branch for integers is new to the version, but its implementation simply makes use of a InforML primitive for converting integers into strings.

Finally, this new version of `toString` provides a wildcard case. This case uses `isdata` to check whether the input is an algebraic data type. If so, it use `toSpine` to convert the argument `x` into a `Spine`. It also uses a new InforML primitive that I have not yet covered, `stringDatacons`, to create a string to pass off to `spineToString` along with the `Spine`. If the input is not an algebraic data type, `toString` returns the string `"<Unknown data>"` to indicate that it encountered in input it cannot handle.

The function `stringDatacons` is used to obtain a string name for the data constructor used to build its input value, and has the following type:

```
∀⟨l|α: % @ l|l :> (info α)⟩ α -(⊤)-> String @ l
```

Like `toSpine`, `stringDatacons` will only work on algebraic data types. Its behavior is simple; for example, `stringDatacons (Leaf ⟨⊥⟩)` will evaluate to the value `"Leaf"` and

```
stringDatacons (Node (Leaf ⟨⊥⟩) (Leaf ⟨⊥⟩))
```

will evaluate to the value `"Node"`.

Most of the complexity in this extended version of `toString` has been placed into the helper function `spineToString`. This function walks down a `Spine`, calling `toString` to convert the arguments found in `SCons` and `SConsEx` to strings, and returns the string that is the name of the head data constructor when reaching the `SHead` node.

It is worth noting that the reason that the string for the `SHead` is passed as an argument to `spineToString` rather than computing it from the argument of `SHead`, is because the way the InforML compiler chooses to represent data constructors. The consequence of this implementation peculiarity is that the name of data constructor is much easier to obtain when it is fully applied rather than from the data constructor value directly.

Most of the complexity in the definition of `spineToString` comes from needing to specify the type instantiations for calls to `toString` and recursive calls to itself. InforML's local type inference algorithms are not sophisticated enough to determine the correct instantiations from `spineToString` and `toString`'s arguments.

The case for the `SConsEx` function makes use of the `ifholds` primitive to check whether the label hidden by `SConsEx` is less than the requested information content, `l`, of the output string. If so, it will

11. The example, however, still does not handle arbitrary tuples. Handling n-ary tuples, for all n, requires another primitive function, similar to `toSpine`, but less interesting.

```
module tdp : sig
  fun toString : ∀⟨l|α: * @ l|l = (info α)⟩ α -(l)-> String @ l
end = mod
  fun toString : ∀⟨l|α: * @ l|l = (info α)⟩ α -(l)-> String @ l

  fun spineToString : ∀⟨l|α: * @ l|l = (info α)⟩
                       (| String @ l, Spine α @ l |) -(l)-> String @ l

  fun toString ⟨l|α⟩ arg =
      typecase α
             | Int @ _ => stringInt arg
             | String @ _ => "\"" ^ arg ^ "\""
             | Bool @ _ => if arg then "True" else "False" end
             | _ -(_ | _)-> _ => "<Function>"
             | (β, ψ)      =>
               "(" ^ (toString ⟨l|β⟩ (arg.0)) ^ "," ^ (toString ⟨l|ψ⟩ (arg.1)) ^ ")"
             | _ => isdata α then
                      spineToString ⟨l|α⟩
                        (stringDatacons ⟨l|α⟩ arg)
                        (toSpine ⟨l|α⟩ arg)
                    else
                      "<Unknown data>"
                    end
          end

  and spineToString ⟨l|α⟩ name spn =
      case spn
        of SHead _ => name
         | SCons ⟨l -- |ω ψ⟩ newspn arg =>
             (spineToString ⟨l|(ω -(l|l)-> ψ)⟩ name newspn) ^
             " (" ^ (toString ⟨l|ω⟩ arg) ^ ")"
         | SConsEx ⟨l l' -- |ω ψ⟩ newspn arg =>
             ifholds l' <: l then
               ifholds (info ω) = l then
                 (spineToString ⟨l|(ω -(l|l)-> ψ)⟩ name newspn) ^
                 " (" ^ (toString ⟨l|ω⟩ arg) ^ ")"
               else
                 (spineToString ⟨l|(ω -(l|l)-> ψ)⟩ name newspn) ^
                 " <Redacted>"
               end
             else
               "<Redacted>"
             end
       end
```

Figure 3·5: A complete version of `toString` in InforML.

then check whether the information content of `ω` is equal to the requested information content `l`. If not, it will recursively call `spineToString` on the remainder of the `Spine` and use the string `"<Redacted">` for the argument. Otherwise it will call `toString` recursively on the argument. If the hidden label was not less than `l`, `spineToString` simply returns the string `"<Redacted>"` and makes no further recursive calls.

It may seem too conservative to fail to print the remainder of a `Spine` if the label `l'` is not less than the label `l`. However, the problem arises from the fact that to call `spineToString` recursively, it must instantiate its type argument `α` with the type `ω -(l|l)-> δ`. However, because the information content of the type `ω -(l|l)-> δ` is computed from the combination of `ω`'s information content, `l'`, `δ`'s information content (`l`), and the current program counter (`l`). If the label `l'` is greater than `l` then the information content of the entire type will be greater than `l`. Consequently, the value produced by a recursive call to `spineToString` will not have an information content less than or equal to `l`, contradicting `spineToString`'s type signature.

This restriction is annoying because the recursive call to `spineToString` does not need to make use of the type `ω`. I conjecture that this problem could be solved if a "bottom" type were added to InforML. That way it would be possible to instantiate the recursive call with a type like `Bot -(l|l)-> δ` instead of `ω -(l|l)-> δ`.

Now that I have finished my explanation of programming in InforML, I will conclude this chapter by describing how InforML relates to other functional languages and languages with information-flow type systems.

§ 3·6 Related work

InforML builds on most of the functional languages that have come before it. InforML is most directly descended from AspectML (Dantas, Walker, Washburn, and Weirich 2008), and was built from its code base. However, over the past year InforML's evolution has diverged greatly. Aside from syntactic differences, InforML no longer supports global type inference like AspectML, lacks support for stack analysis, and its aspect-oriented features have been simplified and refined.[12] On the other hand, InforML extends AspectML with an information-flow type and kind system, type functions, a second-class module system, and first-class existential types.

InforML and AspectML are both, in turn, descended from Standard ML (Milner et al. 1997), Objective Caml (Leroy et al. 2000), and Haskell (Peyton Jones 2003). The relationship with Standard ML and Objective Caml is, however, mostly syntactic. With the exception of InforML's module system, and the fact that InforML and AspectML both use a call-by-value operational semantics, most of their advanced type systems features are closer to what would be found in a modern Haskell implementation rather than a modern ML-like language: polymorphic recursion (Mycroft 1984), existential algebraic data types (Läfer and Odersky 1994), higher-kinded polymorphism, higher-rank polymorphism (Peyton Jones et al.), GADTs (Coquand 1992; Crary and Weirich 1999; Xi, Chen, and Chen 2003; Peyton Jones, Vytiniotis, Weirich, and Washburn 2006), constrained polymorphism, etc.

12. I have not discussed aspect-oriented programming in InforML, because it is mostly orthogonal to the central theme of this dissertation.

To date, other than InforML, there are only two other realistic implementations of programming languages with an information-flow type system: Jif (Chong et al.) and FlowCaml (Simonet 2003). While Jif compiles to Java byte-code, it does not provide wrappers for Java's reflection library. Like Jif, FlowCaml does not provide any mechanisms for TDP. Recent versions of Jif have added support for runtime principal and label analysis, however. Runtime principal analysis was first formally studied in the work by Tse and Zdancewic (2004a; 2005). Furthermore, because principles and labels are defined as part of their language of types, it can be seen a restricted form of runtime type analysis. However, Tse and Zdancewic did not consider issues of type abstraction in their proofs of noninterference. Concurrently with Tse and Zdancewic's research, Zheng and Myers (Zheng and Myers 2004) developed a formalization of dynamic labels and label analysis in Jif. Their label analysis primitive is quite similar to the `ifholds` primitive provided by InforML. Furthermore, Zheng and Myers resolved the problem of using existentially quantified labels as a covert channel by reifying labels as values. These values representing labels have a type of "label" which is itself labeled.

I will focus on comparing InforML with FlowCaml because they are both ML-like languages, while Jif is based upon the Java (Gosling et al. 2005) language. The current version of Jif lacks support for Java-style generics, but does offer label and principal polymorphism. Therefore, Jif primarily relies on subtype polymorphism instead of parametric polymorphism. InforML and FlowCaml, on the other-hand both use structural subtyping induced by the ordering on labels. FlowCaml and InforML do not have a mechanism for introducing nominal subtypes, so programs written in FlowCaml and InforML rely on parametric polymorphism. Therefore, it is possible to make a more detailed comparison between a program written in InforML and the same program written in FlowCaml.

InforML mostly subsumes FlowCaml in terms of functionality. FlowCaml does implement global type inference, module functions (functors), exceptions, and provides a novel type visualization tool. On the other-hand, FlowCaml does not have an information-flow kind system, has no support for type-directed programming, does not provide existential data types, GADTs, higher-rank polymorphism, higher-kinded polymorphism, label analysis, or first-class existentials.

For the features that InforML and FlowCaml have in common, there are a number of small differences. Like InforML, FlowCaml gives types and labels distinct kinds, called `type` and `level`, but does not differentiate them as strictly syntactically. This can be illustrated in FlowCaml interactive top-level:

```
# let x = 1;;
x : 'a int
#
```

FlowCaml uses prefix notation for type and label arguments, here `'a` is the label describing the information content of the integer. All universal quantification in FlowCaml is implicit so this type should be understood as `∀'a:level. 'a int`. The same interaction with InforML will look like:

```
- val x = 1;
val x = 1 : Int @ ⊥
-
```

The two primary differences here are that InforML will infer the label ⊥ instead of universally quantifying over the label, and that label application must be distinguished from type application in InforML. Universal

quantifiers can only be introduced in InforML through function or data constructor definitions. Therefore, local type inference in InforML will not introduce a universal quantifier.

In InforML, applying a type function to a type and applying a type function to a label have a distinct syntax to enforce a grammatical distinction between types and labels. The reason for this distinction is that I wanted to prevent programmers from being able to write types like

```
Int @ ((λ l:Lab =(+)=> l ⊔ ⊤ end) @ ⊥).
```

If it were possible to mix types and labels in this manner, it would greatly complicate the algorithm for solving label constraints. Furthermore, I believe the distinction makes types easier to read. The type of lists in InforML is written `List α @ l`, which makes it easy to distinguish which is the type of the elements and what is the information content of the entire list. In FlowCaml the type type of lists is written `('a, 'b) list` where `'b` is the information content of a list.

A small difference between FlowCaml and InforML is in their function types. The function type

```
∀⟨l1 l2 l3 l4⟩ Int @ l1 -(l2|l3)-> Int @ l4
```

in InforML would be written in FlowCaml as

```
'l1 int -{'l2||'l3}-> 'l4 int
```

in FlowCaml. The first field in the FlowCaml function type, `'l2`, is the program counter, the second field is an empty *row type* (Wand 1987, 1988; Rémy 1990) that is used to track information concerning exceptions, and the final field, `'l3` is the information content of the function closure itself.

As I mentioned § 3·1, `info` labels in InforML were partly inspired by level constraints in FlowCaml. A level constraint in FlowCaml is written as `'b < level('a)` and means that the information content of any type used to instantiate `'a` must be greater than or equal to the label `'b`. In most cases, level constraints behave identically to `info` labels. For example, the following InforML function:

```
fun choose : ∀⟨l|α: * @ ⊥|l <: (info α)⟩ α -(⊥)-> α -(⊥)-> Bool @ l -(l)-> α
```

when written in FlowCaml:

```
let choose y1 y0 x = if x then y1 else y0
```

will be inferred to have the type `'a -> 'a -> 'b bool -> 'a with 'b < level('a)`. In FlowCaml constraints are written follow a type, rather than preceding it. Note that < is used to mean less-than or equal in FlowCaml constraints.

However, level constraints *are* constraints while `info` labels *are* labels. Therefore, `info` labels can be used to instantiate label polymorphic functions. However, arguably the only times when this capability would be useful arise because InforML does not perform global type inference.

In addition to level constraints, FlowCaml also has two other forms of constraint that InforML lacks: content constraints and skeleton constraints. Contents constraints are written as `content('a) < 'b` and mean that the information content of the type `'a` and *every* one of its subterms must be less than or equal to the label `'b`. This constraint is motivated by FlowCaml's polymorphic comparison functions. Something like content constraints could be very useful for writing type-directed functions in InforML.

However, because FlowCaml does not allow for existentially quantified types and labels, it is very easy to check content constraints by structural recursion on their type argument. I do not believe that there is any way to implement such a constraint in InforML because it allows existentially quantified types and labels.

Skeleton constraints are written as `'a ~ 'b` and means that the *skeleton* of the type `'a` must match the *skeleton* of type `'b`. The skeleton of a type in FlowCaml is the structure of a type, ignoring labels. This constraint would again be very useful in InforML. For example, the following FlowCaml is the identity on values, but coerces the label so that output has an information that is greater than or equal to the input:

```
(* increaseLevel has type 'a -> 'b
                     with 'l1 < level('a)
                     and  'l2 < level('b)
                     and  'l1 < 'l2
                     and 'a ~ 'b *)
let increaseLevel (x : 'a) : 'b = x
```

I could try to write this function in InforML as

```
fun increaseLabel : ∀⟨l1 l2|(α: * @ ⊥) (β: * @ ⊥)|l1 <: (info α) &
                                                  l2 <: (info β) &
                                                  l1 <: l2⟩ α -(⊥)-> β
fun increaseLabel ⟨l1 l2|α β⟩ (x : α) : β = x
```

but it will fail to typecheck because there is no reason that α and β should have any relationship. Rewriting the function as the following:

```
fun increaseLabel : ∀⟨l1 l2|(α: * @ ⊥)|l1 <: (info α) &
                                       l2 <: (info α) &
                                       l1 <: l2⟩ α -(⊥)-> α
fun increaseLabel ⟨l1 l2|α⟩ (x : α) : α = x
```

will allow it to typecheck, but has different meaning. For example, it would be possible to instantiate the FlowCaml function `increaseLevel` to give it the type `'l1 int -> 'l2 int with 'l1 < 'l2`, but it is not possible to instantiate and coerce the InforML version, `increaseLabel`, so that it can be used with the type ∀⟨l1 l2l1 <: l2⟩ Int @ l1 -(⊥)-> Int @ l2|. The function with the closest meaning in InforML to `increaseLevel` in FlowCaml would in InforML is the following:

```
fun increaseLabel : ∀⟨l1 l2|α: Lab -(+)-> * @ ⊥|l1 <: l2⟩ α @ l1 -(⊥)-> α @ l2
fun increaseLabel ⟨l1 l2|α⟩ (x : α @ l1) : α @ l2 = x
```

Because it is not always desirable to quantify over functions from labels to types, it may be reasonable to incorporate a feature like skeleton constraints in a future version of InforML.

Finally, while both InforML and FlowCaml allow the programmer to write variance annotations for the arguments to algebraic data types, FlowCaml provides a *guard* annotation in addition to co-, contra-, and invariant annotations. Written as #, this variance indicates that the argument is covariant and specifies the information content of the algebraic data type. For example, where the option data type is defined as

```
datatype Option : Π l:Lab -(+)-> (* @ l) -(+)-> Lab -(+)-> (% @ ⊥) =
   | None : ∀⟨l1 l2|α: * @ l1⟩ Option α @ l2
   | Some : ∀⟨l1 l2|α: * @ l1|(info α) = l2⟩ α -(⊤)-> Option α @ l2
```

in InforML, it would be defined in FlowCaml like

```
type ('a, 'b) option =
     None
   | Some of 'a
   # 'b
```

and would be inferred by the FlowCaml type inference algorithm to mean

```
type (+'a:type, #'b:level) option = None | Some of 'a # 'b
```

As I have discussed, InforML just uses the convention that the last label argument in a type is the information content of the corresponding value. Therefore, InforML can be viewed as requiring the kind of every algebraic data type to end in the kind `Lab -(#)-> (% @ l)`.

There are a few reasons why FlowCaml's definition of `option` is much shorter than InforML's definition of `Option`. Partly this is because FlowCaml does not require a kind annotation when defining new algebraic data types. Additionally, InforML uses a verbose syntax for defining GADTs, styled after Haskell's syntax for defining GADTs (Peyton Jones, Vytiniotis, Weirich, and Washburn 2006), and for simplicity I chose to make that the only way to define algebraic data types, rather than having a separate syntax for defining standard algebraic data types.

Additionally, the constraint on the information content of `Some`'s type argument, that `(info α) = l2` is not strictly necessary for regular programming. However, for TDP the constraint proves fairly useful. Consider the example, I gave in § 3·3 of extending `toString` with a case for `Option`:

```
fun toString : ∀⟨l|α: * @ l|(info α) = l⟩ α -(l)-> String @ l
fun toString ⟨l|α⟩ arg =
    typecase α
           ...
           | Option @ l1 β @ l =>
             case arg
                | Some arg' =>
                  "Some " ^ (toString ⟨l|β⟩ arg')
                | None      => "None"
              end
         end
```

If `Some` did not have this constraint inside the `case` branch above, nothing would be known about the information content of β. However, the program counter requires that all values have an information content of at least `l`. If nothing is known about the information content of β, there is no way to show that expression `arg'` in the code above is well-typed. Consequently, without the constraint annotation on `Some`, the above program will be rejected.

4

InforML information-flow usage patterns

The mark of our time is its revulsion against imposed patterns.
Marshall McLuhan (*Understanding Media: The Extensions of Man*, 1994)

"Mike, I can't believe you brought the Taint into our office."
Douglas Coupland (*jPod*, 2006)

In the previous chapter, I introduced the language InforML which uses an information-flow type and kind system to provide programmers with static guarantees about the confidentiality and integrity of their abstract data types. In this chapter I will describe three usage patterns for InforML and the static guarantees InforML provides when using these patterns.

I begin by describing *intra-module type-directed programming*, whereby the author of a module can use TDP as part of implementing their module regardless of the policy that they specify for type-directed operations written outside the module. I then move onto explaining the *harmless reflection* pattern, which gives a module the strongest possible static reasoning principles for confidentiality and integrity. I then present the *break and recover* pattern, which provides a weaker alternative to harmless reflection where confidentiality is not guaranteed to hold but integrity is preserved.

Between discussing the harmless reflection idiom and the break and recover idiom, I will introduce the idea of using wrapper types to make the information-flow constraints from using `typecase` with generative data types less conservative.

§ 4·1 Intra-module type-directed programming

As a running example in this chapter, I will be using an InforML implementation of the data structure for companies from § 1·2. An initial translation of the code is presented in Figure 4·1. This implementation,

```
module companies = mod

  type name    = String
  type address = String

  datatype Person : Lab -(+)-> % @ ⊥ =
            | P : ∀⟨l⟩ (| name @ l, address @ l|) -(⊤)-> Person @ l

  datatype Salary : Lab -(+)-> % @ ⊥ =
            | S : ∀⟨l⟩ Int @ l -(⊤)-> Salary @ l

  datatype Employee : Lab -(+)-> % @ ⊥ =
            | E : ∀⟨l⟩ (| Person @ l, Salary @ l|) -(⊤)-> Employee @ l

  type manager = Employee

  datatype Dept : Lab -(+)-> % @ ⊥  =
            | D : ∀⟨l⟩ (| name @ l,
                          manager @ l,
                          Int @ l,
                          List (SubUnit @ l) @ l |) -(⊤)-> Dept @ l

       and SubUnit : Lab -(+)-> % @ ⊥ =
            | PU : ∀⟨l⟩ Employee @ l -(⊤)-> SubUnit @ l
            | DU : ∀⟨l⟩ Dept @ l -(⊤)-> SubUnit @ l

  datatype Company : Lab -(+)-> % @ ⊥ =
            | C : ∀⟨l⟩ List (Dept @ l) @ l -(⊤)-> Company @ l
end
```

Figure 4·1: An InforML implementation of a module for companies.

as suggested there, adds a field to the `Dept` algebraic data type to cache valuations. I have defined all of the data constructors to be label polymorphic in order to maximize their reuse, as well as more precisely track information flows.

The `Department` and `Company` algebraic data types both make use of the type `List` from the InforML standard library. The `List` algebraic data type is defined as follows:

```
   datatype List : Π l:Lab -(+)-> * @ l -(+)-> Lab -(+)-> % @ l =
      | Nil : ∀⟨l1 l2|α: * @ l1⟩ List α @ l2
      | ::  : ∀⟨l1 l2|α: * @ l1|(info α) = l2⟩ (| α, List α @ l2 |) -(⊤)-> List α @ l2
```

The data constructor `Nil` is the empty list and the infix data constructor `::` is list concatenation. InforML supports the syntactic sugar for lists as, $[e_1, \ldots, e_n]$, like in Standard ML.

I could re-implement the valuation functions (`valCompany`, …, `valSalary`) provided by the module for companies in § 1·2, using the usual recursive pattern matching boilerplate, but the goal of this section

```
fun valuation : ∀⟨l|α: * @ l|(info α) = l⟩ α -(l)-> Int @ l
fun valuationSpine : ∀⟨l|α: * @ l|(info α) = l⟩ Spine α @ l -(l)-> Int @ l

fun valuation ⟨l|α⟩ arg =
      typecase α
             | Salary @ l => case arg of S i => i end
             | Dept @ l => case arg of D _ _ vl _ => vl end
             | _ => isdata α then
                      valuationSpine ⟨l|α⟩ (toSpine ⟨l|α⟩ arg)
                    else
                      0
                    end
          end

and valuationSpine ⟨l|α⟩ spn =
    case spn
       | SHead _ => 0
       | SCons ⟨l1 l2|β ψ⟩ newspn arg =>
           (valuationSpine ⟨l2|(β -(⊤|l)-> ψ)⟩ newspn) + (valuation ⟨l2|β⟩ arg)
       | SConsEx _ _ =>
           abort "Impossible"
     end
```

Figure 4·2: A type-directed implementation of a valuation function.

is to show how the author of a module can take advantage of TDP inside of her module, regardless of the policy for code written outside to the module. The author, of course, is bound by the policies defined by any modules that she uses as part of her implementation. However, I can be certain that for the `companies` module that there are no restrictions on the use of TDP within the module because it does not depend upon any non-primitive types defined outside the module.

One possible type-directed implementation of a unified valuation function is given in Figure 4·2. The definitions of `valuation` and `valuationSpine` are mostly straightforward. If `valuation` is given a value of type `Salary` it will use pattern matching to extract the value. If `valuation` is given a value of type `Dept`, it will return its cached valuation rather than make a recursive call. Finally, for all other inputs, `valuation` checks whether the value is an algebraic data type. If so, `valuation` converts it to a `Spine` and calls `valuationSpine`, which will walk down the spine and recursively call itself and `valuation`. If the argument is not an algebraic data type, `valuation` just returns `0`.

The only unusual part is that `valuationSpine` will abort when given a `SConsEx` constructor. This behavior is justified by the definitions of the data constructors in the `companies` module. It is a property of `toSpine` that it will never create an `SConsEx` constructor when called on values that only contain labels less than the information content of the value, as per Proposition 4·1·2 and Corollary 4·1·3 below.

Definition 4·1·1 (`SConsEx` free)**.** *I will call a value* `SConsEx` *free, if the value, and all of its subterms, are not the* `SConsEx` *data constructor.*

```
sig
  type name : Lab -(+)-> * @ ⊥ = String
  type address : Lab -(+)-> * @ ⊥ = String

  data Person : Lab -(+)-> % @ ⊥
  con P : ∀⟨l⟩ (| name @ l, address @ l|) -(⊤)-> Person @ l
  data Salary : Lab -(+)-> % @ ⊥
  con S : ∀⟨l⟩ Int @ l -(⊤)-> Salary @ l

  data Employee : Lab -(+)-> % @ ⊥
  con E : ∀⟨l⟩ Person @ l -(⊤|⊥)-> Salary @ l -(⊤)-> Employee @ l

  type manager : Lab -(+)-> % @ ⊥ = Employee

  data Dept : Lab -(+)-> % @ ⊥
  data SubUnit : Lab -(+)-> % @ ⊥
  con D : ∀⟨l⟩ name @ l -(⊤)->
                Employee @ l -(⊤)->
                Int @ l -(⊤)->
                List (SubUnit @ l) @ l -(⊤)-> Dept @ l

  con PU : ∀⟨l⟩ Employee @ l -(⊤)-> SubUnit @ l
  con DU : ∀⟨l⟩ Dept @ l -(⊤)-> SubUnit @ l

  data Company : Lab -(+)-> % @ ⊥
  con C : ∀⟨l⟩ List (Dept @ l) @ l -(⊤)-> Company @ l

  fun valuation : ∀⟨l|α: * @ ⊥|(info α) = l⟩ α -(l)-> Int @ l
  fun valuationSpine : ∀⟨l|α: * @ ⊥|(info α) = l⟩ Spine α @ l -(l)-> Int @ l
end
```

Figure 4·3: The inferred signature for the `companies` module.

Proposition 4·1·2 (Properties of `toSpine`)**.**

When `toSpine ⟨l⟩` *is applied to some value,* `D` $\mathtt{v}_1 \ldots \mathtt{v}_n$*, it will produce a result that is* `SConsEx` *free iff*

- *the information content of the value's arguments,* $\mathtt{v}_1$ *through* $\mathtt{v}_n$*, are all less than or equal to* `l`*,*
- *the information content of the value's argument's types,* τ_1 *through* τ_n*, are all less than or equal to* `l`*,*

Corollary 4·1·3 (Instantiating `toSpine` with ⊤)**.** *If* `toSpine` *is instantiated with the label* ⊤*, the resulting* `Spine` *will always be* `SConsEx` *free.*

Proof. Follows as a consequence of Proposition 4·1·2 when `l` is ⊤. □

The only disadvantage of the above implementation of `valuation` is that its type signature allows it to be applied to any value, not just values constructed from the data constructors in the `companies`

module. A more precise type for `valuation`, than the one in given in Figure 4·2, could be written by introducing a common supertype for all of the algebraic data types in the `companies` module.

Even though InforML cannot define generative types to be subtypes of existing types, there are many ways, such as merging all of the algebraic datatypes or using phantom types (Fluet and Pucella 2006), to simulate this kind of subsumption in a language like InforML. However, calling `valuation` on other sorts of values does not impair the operation of the `companies` module. So, at present, I will not complicate my presentation by using one of these techniques, but I will return to this idea in § 4·3.

Figure 4·3 shows the signature that InforML would infer for the `companies` module. This signature does not provide any guarantees concerning confidentiality and integrity.

- Any changes to the implementation can affect any code that uses the `companies` module, and the type system not provide any indication of which parts of a program can depend on this implementation. Therefore, none of the type definitions or algebraic data types have the confidentiality property.
- Additionally, there is nothing stopping code in the program from creating an invalid instance of the `Dept` algebraic data type, where the cached valuation is equal to the sum of valuation of its manager and `SubUnits`. Therefore, none of these type definitions and algebraic data types have the integrity property.

As an example of how confidentiality can be broken, here is the code of the InforML version of the `increase` function that was discussed in § 1·2:

```
fun increase : ∀⟨l|α: * @ l|(info α) = l⟩ (| α, Int @ l |) -(l)-> α
fun increaseSpine : ∀⟨l|α: * @ l|(info α) = l⟩ (| Spine α @ l, Int @ l |) -(l)-> α

fun increase ⟨l|α⟩ arg amt =
      typecase α
             | companies.Salary @ l =>
                case arg
                   | companies.S ⟨l1⟩ i => companies.S ((i * amt) div 100)
                 end
             | _ => isdata α then
                      increaseSpine ⟨l|α⟩ (toSpine ⟨l|α⟩ arg) amt
                    else
                      arg
                    end
          end

and increaseSpine ⟨l|α⟩ spn amt =
    case spn
       | SHead dc => dc
       | SCons ⟨l1 l2|β ψ⟩ newspn arg =>
           (increaseSpine ⟨l2|(β -(⊤|l)-> ψ)⟩ newspn amt) (increase ⟨l|β⟩ arg amt)
       | SConsEx _ _ =>
           abort "Impossible"
```

The structure of `increase` is very similar to `valuation`. The primary difference is that `increase` returns a value with the same type of its input.

Having introduced the `companies` module as my starting point, which has no static guarantees concerning the confidentiality and integrity, I will now proceed to show how three different idioms can be applied to this example and the properties that can be derived from them.

§ 4·2 Harmless reflection

This idiom takes its name in analogy to *harmless advice* (Dantas 2007) in aspect-oriented programming (Kiczales et al. 2001; Dantas et al. 2008). Harmless advice is designed so that when the advice is woven into a program, it will not affect the behavior of the original program. I will call this original behavior the *essential computations* and the new computations performed by the harmless advice *inessential computations*. Similarly, with harmless reflection all uses of TDP that could break confidentiality and integrity are disallowed from affecting the essential computations.

Essential computations are those that directly contribute to the "goal" of a program, such as producing the expected output value. Inessential computations are those that may occur, but are either independent from or do not contribute directly to the primary goal of the program. For example, the primary goal of a web server is to accept requests from the network and provide responses. Usually a web server will also log transactions and debugging output. However, how the web server handles requests and responses is (usually) entirely independent of what it writes to its log files. Therefore, if a web server were written using the harmless reflection idiom, it would be designed to ensure that any uses of TDP that break confidentiality will not affect the essential computations, those that involve requests and responses. However, it would be probably be acceptable for computations that are part of the logging infrastructure to break confidentiality because they will not affect the essential behavior the program.

It is, of course, left to the programmers to decide which computations are essential and which computations are inessential. What InforML does is allow programmers to use its type system to clearly delineate the two and enforce the high-level policies they choose. Furthermore, InforML will help programmers in partitioning their programs. If a programmer tries to make an essential computation depend on an inessential computation, InforML will report a type error identifying the mistake.

When programming in the harmless reflection idiom, all term data belonging to essential computations should be declared public knowledge, that is labeled with $\bot$. All types that are intended to be fully abstract, that is, protected from potentially harmful uses of reflection, should be placed inside of modules and ascribed with a signature that gives them a maximally restricted information content, in other words, $\top$. In this idiom, it is possible to use TDP to help define a module because the type definitions and algebraic data types are all public knowledge inside the module. Outside the module, using TDP will result in data with a restricted information content. However, because the essential computations in the program have been given types that only accept inputs that are public knowledge, the data created by breaking representation independence can never be used as part of these essential computations. Therefore, changing the implementation of a fully abstract data type will never affect the behavior of the essential computations.

So how can harmless reflection be applied to the `companies` module? First, it is necessary to define a signature to make the definitions of its algebraic data types abstract.

```
signature companies = sig
  type name     : Lab -(+)-> * @ ⊥ = String
  type address  : Lab -(+)-> * @ ⊥ = String
  data Person   : Lab -(+)-> % @ ⊤
  data Salary   : Lab -(+)-> % @ ⊤
  data Employee : Lab -(+)-> % @ ⊤
  type manager  : Lab -(+)-> % @ ⊤ = Employee
  data Dept     : Lab -(+)-> % @ ⊤
  data SubUnit  : Lab -(+)-> % @ ⊤
  data Company  : Lab -(+)-> % @ ⊤
```

In this signature all of the algebraic data types have been given a kind that construct types with the restricted information content, ⊤. The signature exposes the definitions of `name`, `manager`, and `address`, but hides all of the data constructors in the module. If the signature exposed the data constructors, the algebraic data types would not be abstract – anyone could just pattern match on their values. Consequently, there is a need to define helper functions in the `companies` module so that it is possible to construct and destruct values of the abstract algebraic data types.

However, it is worth noting that it is not possible to just use signature ascription to prevent data constructors from being used. It might seem reasonable, for example, to allow the following signature:

```
sig
   data T : Lab -> * @ ⊥
   con MkT : T @ ⊥ end
end
```

to be subsumed by the signature:

```
sig
   data T : Lab -> * @ ⊥
   val MkT : T @ ⊥ end
end
```

because exposing `MkT` as a value rather than a constructor would prevent it from being used for pattern matching. The reason for this is rather mundane: `MkT` is not a lexically valid variable name in InforML, and therefore cannot be used to name a value.

The definitions for these helper functions, given in Figure 4·4, are mostly tedious. The only interesting bit is the definition of the `newDept` function. Because the `Dept` algebraic data type caches its valuation, when constructing a new value of this type it is necessary to pre-compute its valuation.

There also proves to be a difficulty with the `valuation` function. We would like the signature to expose it with the type:

```
signature COMPANIES = sig
  ...
  fun valuation : ∀⟨α: * @ ⊥|(info α) = ⊥⟩ α -(⊥)-> Int @ ⊥
  ...
end
```

```
module companies = mod
  ...
  # Constructors
  fun newCompany : List (Dept @ ⊥) @ ⊥ -(⊥)-> Company @ ⊥
  fun newCompany ds = C ds

  fun newDept : (| name @ ⊥, manager @ ⊥, List (SubUnit @ ⊥) @ ⊥ |) -(⊥)-> Dept @ ⊥
  fun newDept nm mn sbs =
      D nm mn ((valuation ⟨⊥|(manager @ ⊥)⟩ mn) +
               (valuation ⟨⊥|(List (SubUnit @ ⊥) @ ⊥)⟩ sbs)) sbs
  ...

  # Accessors
  fun companyDepts : Company @ ⊥ -(⊥)-> List (Dept @ ⊥) @ ⊥
  fun companyDepts (C ds) = ds

  fun deptName : Dept @ ⊥ -(⊥|⊥)-> name @ ⊥
  fun deptName (D nm _ _ _) = nm
  ...
```

Figure 4·4: Helper functions for the `companies` module.

But now that all of the algebraic data types in the `companies` module have been given the restricted information content, ⊤, the function `valuation` cannot be used on their data constructors. For example, it is not possible to call valuation on a value of type `Company @ ⊥`, because `Company @ ⊥` has kind `% @ ⊤`. But the new signature for `valuation` can only be instantiated with types of kind `* @ ⊥`, and the kind `% @ ⊤` is not subkind of `* @ ⊥`. Changing the signature does not solve the problem either:

```
signature COMPANIES = sig
  ...
  fun valuation : ∀⟨α: * @ ⊤|(info α) = ⊥⟩ α -(⊥)-> Int @ ⊥
  ...
end
```

This signature is invalid because `∀⟨l|α: * @ ⊥|(info α) = l⟩ α -(l)-> Int @ l` is not a subtype of `∀⟨α: * @ ⊤|(info α)= ⊥⟩ α -(l)-> Int @ ⊥`. Again this problem could be resolved by using phantom types, or related techniques, to simulate a common supertype, but for the present I will assume that it is only ever necessary to calculate the valuation of an entire company:

```
signature COMPANIES = sig
  ...
  fun valuation : Company @ ⊥ -(⊥)-> Int @ ⊥
  ...
end
```

Putting everything together, I can now give the revised implementation of the `companies` module's signature in Figure 4·5.

```
signature companies = sig
  type name     : Lab -(+)-> * @ ⊥ = String
  type address  : Lab -(+)-> * @ ⊥ = String
  data Person   : Lab -(+)-> % @ ⊤
  data Salary   : Lab -(+)-> % @ ⊤
  data Employee : Lab -(+)-> % @ ⊤
  type manager  : Lab -(+)-> % @ ⊤ = Employee
  data Dept     : Lab -(+)-> % @ ⊤
  data SubUnit  : Lab -(+)-> % @ ⊤
  data Company  : Lab -(+)-> % @ ⊤

  # Constructors
  fun newCompany : List (Dept @ ⊥) @ ⊥ -(⊥)-> Company @ ⊥
  fun newDept : (| name @ ⊥, manager @ ⊥, List (SubUnit @ ⊥) @ ⊥ |) -(⊥)-> Dept @ ⊥
  ...

  # Accessors
  fun companyDepts : Company @ ⊥ -(⊥)-> List (Dept @ ⊥) @ ⊥
  fun deptName : Dept @ ⊥ -(⊥)-> name @ ⊥
  ...
  fun valSalary: Salary @ ⊥ -(⊥)-> Int @ ⊥

  # Valuation
  fun valuation : Company @ ⊥ -(⊥)-> Int @ ⊥
```

Figure 4·5: A harmless reflection signature for the `companies` module.

Within the harmless reflection idiom it is possible to make strong claims about the confidentiality of an ADT.

Conjecture 4·2·1 (Confidentiality for harmless reflection)**.** *Any expression,* e*, that violates confidentiality of abstract data types will have a type,* τ*, with a restricted information content,* `(info τ = ⊤`*. Only these expressions, which are necessarily part of inessential computations, can be affected by a change in the implementation of an abstract data type.*

I only state conjectures in this chapter because without a formal metatheory for InforML, it is not possible to be certain that generalized parametricity holds for InforML. Therefore, I am extrapolating from what I know of generalized parametricity in $\lambda_{\text{SEC}i}$. I will discuss the challenges proving generalized parametricity presents in § 6.

Furthermore, it no longer possible to write the `increase` function from the beginning of the chapter. Because the algebraic data types in the `companies` module have been ascribed with an information content of `⊤`, it is necessary to change the kind of `increase`'s type argument. However, inside the definition of `increase` this immediately causes problems.

```
fun increase : ∀⟨l|α: * @ ⊤|(info α) = l⟩ (| α, Int @ l |) -(l)-> α
fun increase ⟨l|α⟩ arg amt =
      typecase α
             | companies.Salary @ _ =>
               companies.newSalary (((companies.valSalary arg) * amt) div 100)
             ...
           end
```

Using `typecase` on α will now raise the program counter label to ⊤. Consequently, when typechecking the term `arg` within the first branch, `arg` will be given the type `companies.Salary @ ⊤`. However, the type signature for `companies` requires that the input to `companies.valSalary` be of type `companies.Salary @ ⊥`. Furthermore, regardless of the type of its input, it is not possible to invoke `companies.valSalary` when the program counter label is ⊤, because the program counter label on `companies.valSalary` is ⊥ and control-flow transfers to code with a program counter label less than the current program counter label are disallowed. Therefore, it becomes impossible to write `increase` (outside of `companies`) in a type-directed fashion.

Furthermore, even if it were somehow possible to write a version of the `increase` function we know by Conjecture 4·2·1 that the value it produces will have an information content of ⊤. However, as can be seen from signature in Figure 4·5, only values with an information content of ⊥ can ever be used.

Conjecture 4·2·2 (Integrity for harmless reflection)**.** *Integrity cannot be violated; any expression,* `e` *of type* `τ` *with a restricted information content,* `(info τ) = ⊤`*, will be unusable as part of essential computations. That is, any essential computation cannot take values with type* `τ` *as inputs.*

Despite the fact that the harmless reflection does give strong static guarantees about confidentiality and integrity, there is something subtly unsatisfactory about the way that signature in Figure 4·5 restricts the use of `typecase`. For example, if I use `typecase` on `companies.Salary` directly, the result of that expression will always be a value with a restricted information content:

```
val (x : Int @ ⊤) =
    typecase companies.Salary @ ⊥
           | companies.Salary @ _ => 0
           | _                    => 1
         end
```

This behavior is, of course, correct given the signature of `companies` and the semantics of InforML. However, it feels unsatisfactory because using `typecase` on a generative data type like `companies.Salary` will never reveal anything about its structure.

Furthermore, it is a common to want to determine whether an abstract type is a specific generative type so that it is possible to use the functions provided by a module to safely work with the algebraic data type rather than using `toSpine`. For example, the following functions is ill-typed:

```
fun getVal : ∀⟨l|α: * @ ⊤|(info α) = l⟩ α -(l|⊥)-> Int @ l
fun getVal ⟨l|α⟩ arg =
    typecase α
           | Int @ _              => arg
           | companies.Salary @ _ =>
               companies.valSalary arg
           | _ =>
               abort "Unexpected input"
         end
```

Intuitively, the confidentiality and integrity of `companies.Salary` will not be violated in this function, but it will fail to typecheck for exactly the same reasons that `increase` fails to typecheck.

Ideally, using `typecase` on algebraic data types, rather than type definitions, should propagate information differently because they are an access control mechanism. Better integrating information-flow and the access control provided by generative data types is an area for future work, and I will discuss the problem at greater length in § 6·1. Fortunately, there are indirect solutions that can be used in the current version of InforML. I will present one such solution in the coming section.

§ 4·3 Analyzing restricted generative types with `typecase`

Returning to the `getVal` function, the problem in writing `getVal` so that it is well-typed stems from the fact that I needed to give α the kind `* @ ⊤` so that it could be called on inputs with the type `Salary`. As a consequence of this change, `typecase` raised the program counter label to `⊤`, making it impossible to call `companies.valSalary`. To prevent this chain of consequences, it is clearly necessary to find some way to write `getVal` so that its type argument can be given an unrestricted information content. Unfortunately, there is no way this can be done with the current definition of the `Salary` algebraic data type.

On the other hand, it is necessary for the `Salary` algebraic data type to have a restricted information content, if I want to follow the harmless reflection idiom. Therefore, the problem cannot be solved by simply changing `getVal` or `Salary`.

The solution I chose is to introduce wrapper algebraic data types, with a low information content, that are GADTs (Coquand 1992; Crary and Weirich 1999; Xi, Chen, and Chen 2003; Peyton Jones, Vytiniotis, Weirich, and Washburn 2006). Below, I have shown how to extend the original `companies` module, defined in Figure 4·1, to use wrappers:

```
module companies = mod
  ...
  datatype T : Π l:Lab -(+)-> (Lab -(+)-> % @ l) -(+)-> Lab -(+)-> % @ ⊥ =
           | Pwrap  : ∀⟨l⟩ Person @ l -(⊤)-> T Person @ l
           | Swrap  : ∀⟨l⟩ Salary @ l -(⊤)-> T Salary @ l
           | Ewrap  : ∀⟨l⟩ Employee @ l -(⊤)-> T Employee @ l
           | Dwrap  : ∀⟨l⟩ Dept @ l -(⊤)-> T Dept @ l
           | SUwrap : ∀⟨l⟩ SubUnit @ l -(⊤)-> T SubUnit @ l
           | Cwrap  : ∀⟨l⟩ Company @ l -(⊤)-> T Company @ l
end
```

The algebraic data type, `T`, is the new wrapper data type. This algebraic data type is unlike any we have seen so far because it is higher-order. Its kind,

```
Π l:Lab -(+)-> (Lab -(+)-> % @ l) -(+)-> Lab -(+)-> % @ ⊥,
```

states that it does not take a type argument but rather a function from labels to restricted algebraic data types. The `T` algebraic data type is then defined to have a data constructor for every one of the original algebraic data types. These data constructors serve as wrappers witnessing, for each algebraic data type `A`, the coercion from values of type `A @ l` to the type `T A @ l`. This is the reason that the wrapper is a GADT: the types of the data constructors are indexed by their arguments.

The type `∀⟨l|α:Lab -(+)-> % @ ⊤⟩ T α @ l` can be viewed as a common supertype for each of the algebraic data types originally defined in the `companies` module. This modification is, in fact, similar to the one I alluded to in beginning of the chapter to help restrict the domain of `valuation`.

It would have also been possible to define `T` as the following:

```
datatype T : Lab -(+)-> % @ ⊥ =
         | Pwrap  : ∀⟨l⟩ Person @ l -(⊤)-> T @ l
         ...
```

However, indexing `T` with a type makes it possible distinguish between `T` and its various instances without needing a value of the type. For example, with the non-indexed version of `T` to determine whether some type α is a salary it would be necessary to write the following code fragment:

```
typecase α
       | T @ _ => case x of Salary _ => ... end
     end
```

This code requires that there is a value of type α, in this case `x`, to determine that α is a salary. With the indexed version it is possible to simply write the following:

```
typecase α
       | T Salary @ _ => ...
     end
```

Here α is only necessary rather than α and a value of type α.

The algebraic data type `T` can also be seen a restricted form of type representation, as I discussed in § 1·4. Pattern matching upon the wrapper data constructors will introduce a refinement, but with `typecase` and `toSpine` in the language, it does not really provide a form of access control. However, it has the benefit of still introducing type refinements.

It is now also possible to define `valuation` so that its domain is appropriately restricted.

```
fun valuation : ∀⟨l1 l2|α: Lab -(+)-> % @ l1⟩ T α @ l2 -(l2|⊥)-> Int @ l2
fun valuation ⟨l1 l2|α⟩ arg = valuation' ⟨l2|(T α @ l2)⟩ arg
```

Because the original implementation of `valuation` from Figure 4·2, renamed here to `valuation'`, already can calculate the valuation of any type, the most concise implementation for the restricted version of `valuation` is to simply call it to do the real work. It is worth noting that unlike `valuation'`, `valuation` needs to maintain separate labels for its type and term arguments. This is because, inside of the `companies`

module, `valuation` needs to be usable with type `∀⟨α: Lab -(+)-> % @ ⊥⟩ T α @ ⊥ -(⊥)-> Int @ ⊥`, while I wish to expose it with type `∀⟨α: Lab -(+)-> % @ ⊤⟩ T α @ ⊥ -(⊥)-> Int @ ⊥` in the module signature. If I had given `valuation` the type `∀⟨l|α: Lab -(+)-> % @ l⟩ T α @ l -(l)-> Int @ l` it would not be possible to use `valuation` at both these types, because it would not be possible to vary the information content of the type argument and the term argument independently.

The use of wrappers also necessitates rewriting all the helper functions introduced so that the data constructors could be hidden for the harmless reflection idiom.

```
module companies = mod
  ...
  # Constructors
  fun newCompany : List (T Dept @ ⊥) @ ⊥ -(⊥|⊥)-> T Company @ ⊥
  fun newCompany ds =
        Cwrap (C (list.map ⟨⊥ ⊥|(T Dept @ ⊥) (Dept @ ⊥)⟩
                           (λ Dwrap d' => d' end) ds))

  fun newDept : (| name @ ⊥,
                   T Employee @ ⊥,
                   List (T SubUnit @ ⊥) @ ⊥ |) -(⊥|⊥)-> T Dept @ ⊥
  fun newDept nm (Ewrap mn) sbs =
      Dwrap (D nm mn ((valuation' ⟨⊥|(manager @ ⊥)⟩ mn) +
                      (valuation' ⟨⊥|(List (T SubUnit @ ⊥) @ ⊥)⟩ sbs)
             (list.map ⟨⊥ ⊥|(T SubUnit @ ⊥) (SubUnit @ ⊥)⟩
                       (λ SUwrap su' => su' end) sbs))
  ...

  # Accessors
  fun companyDepts : T Company @ ⊥ -(⊥|⊥)-> List (T Dept @ ⊥) @ ⊥
  fun companyDepts (Cwrap (C ds)) =
      list.map ⟨⊥ ⊥|(Dept @ ⊥) (T Dept @ ⊥)⟩ (Dwrap ⟨⊥⟩) ds
  fun deptName : T Dept @ ⊥ -(⊥|⊥)-> name @ ⊥
  fun deptName (Dwrap (D nm _ _ _)) = nm
  ...
```

Each of these constructors and accessors now uses the wrapper constructors to coerce into and out of instances of the algebraic data type `T`. The need for various maps in the revised accessor and constructor functions could be avoided if `T` and the other algebraic data types were defined mutually recursively, but I did not choose to do this because it requires modifying the original data type definitions.

After all these changes to the `companies` module, the revised signature that I will use is given in Figure 4·6.

This revised signature for `companies` includes the new wrapper algebraic data type, `T`, but does not ascribe it a more restrictive kind. Additionally, the signature exposes all of `T`'s data constructors. This is acceptable as they are used as coercions rather than ADTs. Furthermore, it is no longer necessary to write a helper function `valSalary` as it is subsumed by the revised `valuation` function.

```
signature companies = sig
  type name      : Lab -(+)-> * @ ⊥ = String
  type address   : Lab -(+)-> * @ ⊥ = String
  data Person    : Lab -(+)-> % @ ⊤
  data Salary    : Lab -(+)-> % @ ⊤
  data Employee  : Lab -(+)-> % @ ⊤
  type manager   : Lab -(+)-> % @ ⊤ = Employee
  data Dept      : Lab -(+)-> % @ ⊤
  data SubUnit   : Lab -(+)-> % @ ⊤
  data Company   : Lab -(+)-> % @ ⊤

  data T    : Π l:Lab -(+)-> (Lab -(+)-> % @ l) -(+)-> Lab -(+)-> % @ ⊥
  con Pwrap  : ∀⟨l⟩ Person @ l -(⊤)-> T Person @ l
  con Swrap  : ∀⟨l⟩ Salary @ l -(⊤)-> T Salary @ l
  con Ewrap  : ∀⟨l⟩ Employee @ l -(⊤)-> T Employee @ l
  con Dwrap  : ∀⟨l⟩ Dept @ l -(⊤)-> T Dept @ l
  con SUwrap : ∀⟨l⟩ SubUnit @ l -(⊤)-> T SubUnit @ l
  con Cwrap  : ∀⟨l⟩ Company @ l -(⊤)-> T Company @ l

  # Constructors
  fun newCompany : List (T Dept @ ⊥) @ ⊥ -(⊥)-> T Company @ ⊥
  fun newDept : (| name @ ⊥,
                   T manager @ ⊥,
                   List (T SubUnit @ ⊥) @ ⊥ |) -(⊥)-> T Dept @ ⊥
  ...
  fun newSalary : Int @ ⊥ -(⊥)-> T Salary @ ⊥

  # Accessors
  fun companyDepts : T Company @ ⊥ -(⊥)-> List (T Dept @ ⊥) @ ⊥
  fun deptName : T Dept @ ⊥ -(⊥)-> name @ ⊥
  ...

  # Valuation
  fun valuation : ∀⟨α: Lab -(+)-> % @ ⊤⟩ T α @ ⊥ -(⊥)-> Int @ ⊥
```

Figure 4·6: A harmless reflection signature for the wrapped version of the `companies` module.

It is now possible to rewrite the example, `getVal` in the following way:

```
fun getVal : ∀⟨α: * @ ⊥|(info α) = ⊥⟩ α -(⊥|⊥)-> Int @ ⊥
fun getVal ⟨|α⟩ arg =
    typecase α
           | Int @ _                => arg
           | companies.T @ _ _ @ _ =>
               companies.valuation arg
           | _ =>
               abort "Unexpected input"
         end
```

Here, `getVal` can be written so that it can still use `typecase` to determine whether its argument is a "salary", but the program counter will not be raised because the information content of types of the form `T α @ ⊥` have an unrestricted information content. Therefore, it is possible to call `companies.valuation` on `getVal`'s argument.

Furthermore, while it was possible to implement `getVal` and not break confidentiality or produce restricted data, it is still not possible to write `increase` in a type-directed fashion outside of the `companies` module. Unlike before, it is now possible to write the part of `increase` that includes the case for `Salary`. Like `getVal`, `increase` does not need to match beyond the wrapper.

```
fun increase : ∀⟨α: * @ ⊥|(info α) = ⊥⟩ (| α, Int @ ⊥ |) -(⊥)-> α
fun increase ⟨l|α⟩ arg amt =
      typecase α
             | T @ _ companies.Salary @ _ =>
               companies.newSalary (((companies.valSalary arg) * amt) div 100)
             ...
           end
```

However, now the problem in writing `increase` arises in the helper function `increaseSpine`:

```
and increaseSpine ⟨l|α⟩ spn amt =
    case spn
       | SHead dc => dc
       | SCons ⟨l1 l2|β γ⟩ newspn arg =>
           (increaseSpine ⟨l2|(β -(⊤|l)-> γ)⟩ newspn amt) (increase ⟨l|β⟩ arg amt)
       | SConsEx ⟨l1 l2 l3|β γ⟩ newspn arg =>
           (increaseSpine ⟨(l1 ⊔ l2)|(β -(⊤|l)-> γ)⟩ newspn amt)
             (increase ⟨(l2 ⊔ l)|β⟩ arg amt)
     end
```

The problem is that, in order to recursively walk the structure of its input, `increase` must, for some inputs, convert its input into a `Spine` and use a revised version of `increaseSpine`. However, no matter what type I give `increaseSpine` there is no escaping that when it encounters an input that is one of `T`'s data constructors, the information content of the argument, `β`, stored in `SCons` or `SConsEx` is going to be `⊤`. However, to raise the salaries that may occur in the argument `arg`, `increaseSpine` must call `increase`. But `increase` only accepts type arguments with an information content of `⊥`. However, if I change `increase` to accept type arguments with any other information content, it will no longer be

possible to call `companies.valuation`. Consequently, no matter how I push the labels around `increase` will fail to typecheck.

Using wrapped algebraic data types allowed for a more expressive interface, but still retains all of the static guarantees provided by harmless reflection. In the next section, I will discuss the *break and recover* idiom that still preserves integrity, but provides a weaker guarantee concerning confidentiality.

§ 4·4 Break and recover

Because it is always possible to use one of InforML's reflection primitives to analyze the implementation of an ADT, in practice the question is not whether it can occur but whether doing so can be of any consequence. For the harmless reflection idiom I showed that if confidentiality is violated, the information about the structure of the ADT cannot ever impact essential computations. However, sometimes this can be too strong of a restriction for realistic programs. For example, the `increase` function from the previous sections serves a useful purpose. Arguably, `increase` or something like it should be provided as part of the abstraction provided by the `companies` module, but in practice the author of an ADT cannot predict all the operations that could be desirable. Therefore, it can be necessary to write a function like `increase` after the fact, and without access to the ADT's source code.

As I demonstrated for the harmless reflection idiom, it is not possible to implement `increase` using TDP. It can only be written by walking the structure of a `Company` value explicitly by calling the provided accessor functions, and then rebuilding the `Company` using the provided constructor functions.

However, InforML allows the author of an ADT to be a more liberal. She can choose to write a module and signature in such a way that TDP may be used, but still ensure that the integrity property is not violated for her ADTs. This idea is captured by the *break and recover* idiom.[1]

The break and recover idiom works much like the name suggests. Type-directed programming can be used to break the confidentiality property, and like harmless reflection, any data produced will have a tainted information content. Unlike harmless reflection, once data has become tainted it is possible to remove the taint and make the data usable again. By making use of *checked downgrading*, it is possible for the author of an ADT to provide *scrubber*. A scrubber function will verify that tainted values satisfy all invariants internal to the module, and then removes the taint. For some ADTs, it may be reasonable to even allow the *scrubber* to repair invariants that may have been violated. Therefore, the break and recover idiom does not guarantee that the behavior of essential computations is independent of your choice of representation. It does however guarantee that it is not possible to use invalid instances of an ADT.

The break and recover idiom requires exposing the fine structure of InforML labels, which I have been able to avoid discussing so far. The grammar of these refinements can be found in Figure 4·7. In InforML, labels are elements of a free boolean algebra constructed over sets of *atoms*. Atoms are untyped constants. New atoms can be defined in InforML as follows:

```
newatoms a1 ... an
```

1. Though, it might also be called *mostly harmless reflection*.

atomic labels	ℓ	::=	…	
		|	`+{`a_1`, … ,` a_n`}`	additive atom sets
		|	`-{`a_1`, … ,` a_n`}`	subtractive atom sets
declarations	`d`	::=	…	
		|	`newatoms` a_1 … a_n	naming atoms
signature bindings	`sb`	::=	…	
		|	`atom a`	named atom

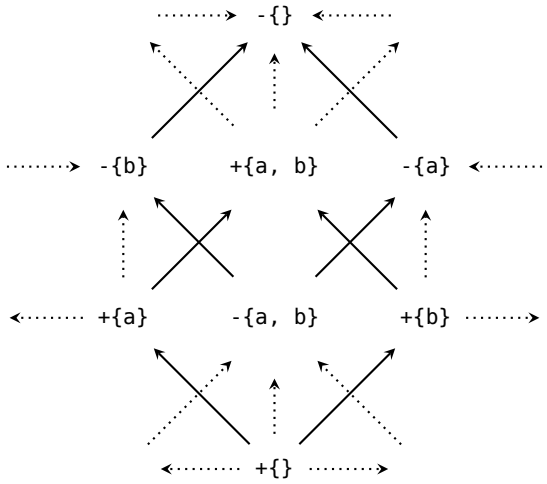
Figure 4·7: The grammar for InforML's fine label structure.

Atoms can also be exported in signature with the following syntax:

```
atom a₁
  …
atom aₙ
```

Because there is assumed to be a countably infinite number of atoms, from a technical standpoint `newatoms` is just providing names for currently unreferenced atoms.

Given named atoms there are two sorts of labels that can be built from them. For example, the atom `a` can be used to construct the label `+{ a }` and the label `-{ a }`. The former label, which I call an *additive set*, denotes the set of atoms containing only the atom `a`. The latter label, which I call a *subtractive set*, denotes the set of all atoms *except* `a`.[2] It is not necessary to have a named atom to use these two label constructors: `+{}` and `-{}` are both valid labels. The former is the empty set of atoms and the latter is the set of all atoms (including those that have yet to be named). These sets are ordered by inclusion so, for just the two atoms `a`, `b`, the following ordering holds:



2. After I had chosen to adopt the names "additive sets" and "subtractive sets", Steve Zdancewic suggested calling them "sets" and "co-sets". I think these names better fit with existing terminology.

$$
\begin{array}{lcl}
\texttt{+\{a}_1\texttt{, ... , a}_n\texttt{\}} \sqcup \texttt{+\{a}'_1\texttt{, ... , a}'_m\texttt{\}} & \triangleq & \texttt{+\{a}_1\texttt{, ... , a}_n\texttt{\}} \cup \texttt{\{a}'_1\texttt{, ... , a}'_m\texttt{\}} \\
\texttt{+\{a}_1\texttt{, ... , a}_n\texttt{\}} \sqcup \texttt{-\{a}'_1\texttt{, ... , a}'_m\texttt{\}} & \triangleq & \texttt{-\{a}'_1\texttt{, ... , a}'_m\texttt{\}} \setminus \texttt{\{a}_1\texttt{, ... , a}_n\texttt{\}} \\
\texttt{-\{a}_1\texttt{, ... , a}_n\texttt{\}} \sqcup \texttt{-\{a}'_1\texttt{, ... , a}'_m\texttt{\}} & \triangleq & \texttt{-\{a}_1\texttt{, ... , a}_n\texttt{\}} \cap \texttt{\{a}'_1\texttt{, ... , a}'_m\texttt{\}} \\
\\
\texttt{+\{a}_1\texttt{, ... , a}_n\texttt{\}} \sqcap \texttt{+\{a}'_1\texttt{, ... , a}'_m\texttt{\}} & \triangleq & \texttt{+\{a}_1\texttt{, ... , a}_n\texttt{\}} \cap \texttt{\{a}'_1\texttt{, ... , a}'_m\texttt{\}} \\
\texttt{+\{a}_1\texttt{, ... , a}_n\texttt{\}} \sqcap \texttt{-\{a}'_1\texttt{, ... , a}'_m\texttt{\}} & \triangleq & \texttt{+\{a}_1\texttt{, ... , a}_n\texttt{\}} \setminus \texttt{\{a}'_1\texttt{, ... , a}'_m\texttt{\}} \\
\texttt{-\{a}_1\texttt{, ... , a}_n\texttt{\}} \sqcap \texttt{-\{a}'_1\texttt{, ... , a}'_m\texttt{\}} & \triangleq & \texttt{-\{a}_1\texttt{, ... , a}_n\texttt{\}} \cup \texttt{\{a}'_1\texttt{, ... , a}'_m\texttt{\}}
\end{array}
$$

Figure 4·8: The definition of label join and label meet for atom set labels.

In the above diagram, arrows point from smaller to larger labels; the dotted arrows are used to emphasize that this is just a small portion of the entire lattice, which has a uncountably infinite number of elements.[3] The ordering on these additive and subtractive sets forms a complete lattice, where `+{}` is the least element and `-{}` is the greatest element. In fact, the label $\bot$ is shorthand in InforML for `+{}`. Dually, $\top$ is shorthand for `-{}`. Figure 4·8 gives the definitions for joins and meets on atom sets.

Having explained the fine structure of labels, I can explain how it is applied in the break and recover programming idiom. Switching to the break and recover idiom from harmless reflection only requires three significant changes to the implementation of a module: at least one new atom should be defined, the accessor and constructor functions for the algebraic data types should be ascribed label polymorphic types, and a scrubber function must be written.

My revised version of the `companies` module uses an atom called `poison`, and I will ascribe the module with the signature given in Figure 4·9. The first thing to note about this signature is that the algebraic data types are now ascribed with kinds that specify that they have an information content of `+{poison}`. Using this label means that it is possible to track those parts of the program that specifically violate the confidentiality of the `companies` module by looking for occurrences of `+{companies.poison}`. In the harmless reflection idiom, by using the $\top$ label, it was not possible to distinguish between the information learned by analyzing different abstract types.

The next change to the `companies` module is to constrain those functions that will only behave correctly if given inputs that meet the invariants of the module. That is, functions that would violate integrity if given an invalid input. In the case of the `companies` module, the `valuation` function is the only function that will behave incorrectly when used with an input that does not meet the invariants of the module. Therefore, the constraint `l <: -{poison}` has been added to the signature for `valuation`. This constraint declares that the function can only used on inputs that are not tainted by the `+{poison}` label. This is similar to the harmless reflection idiom where I chose to ascribe functions with signatures that could only accept data with unrestricted information content, $\bot$. However, this constraint is much weaker because it allow inputs that have been tainted, as long as they have not been tainted by analyzing one of the abstract types in the `companies` module (or by $\top$ which means that it is tainted with respect to every possible ADT).

3. There are $\aleph_0$ atoms, and $2^{\aleph_0}$ additive and subtractive sets, respectively, giving a total of $2^{\aleph_0+1}$ elements, which is equivalent to containing $2^{\aleph_0}$ elements.

```
signature companies = sig
  atom poison
  type name     : Lab -(+)-> * @ ⊥ = String
  type address  : Lab -(+)-> * @ ⊥ = String
  data Person   : Lab -(+)-> % @ +{poison}
  data Salary   : Lab -(+)-> % @ +{poison}
  data Employee : Lab -(+)-> % @ +{poison}
  type manager  : Lab -(+)-> % @ +{poison} = Employee
  data Dept     : Lab -(+)-> % @ +{poison}
  data SubUnit  : Lab -(+)-> % @ +{poison}
  data Company  : Lab -(+)-> % @ +{poison}

  # Constructors
  fun newCompany : ∀⟨l⟩ List (Dept @ l) @ l -(l)-> Company @ l
  ...
  fun newSalary: ∀⟨l⟩ Int @ l -(l)-> Salary @ l

  # Accessors
  fun companyDepts : ∀⟨l⟩ Company @ l -(l)-> List (Dept @ l) @ l
  ...
  fun valSalary : ∀⟨l⟩ Salary @ l -(l)-> Int @ l

  # Valuation
  fun valuation : ∀⟨l|α: * @ l|(info α) = l & l <: -{poison}⟩
                    α -(l)-> Int @ l

  # Scrubbing
  fun scrub : ∀⟨l1 l2|α: Lab -(+)-> % @ l1|l2 <: -{poison} &
                                          l1 <: (l2 ⊔ +{poison})⟩
                α @ l1 -(l2|⊥)-> α @ l2
end
```

Figure 4·9: A break and recover signature for the `companies` module.

Finally, the signature has been extended with a type signature for the scrubber function.

```
fun scrub : ∀⟨l1 l2|α: Lab -(+)-> % @ l1|l2 <: -{poison} &
                                        l1 <: (l2 ⊔ +{poison})⟩
              α @ l1 -(l2)-> α @ l2
```

The scrubber's type signature is fairly complex. It has two label arguments, `l1` and `l2`. Its single type argument, `α`, is a type function from labels to types of base kind with an information content of `l1`. The function itself takes values of type `α @ l1` to values of type `α @ l2`.

The constraints on `scrub` describe the relationship between `l1` and `l2`. The first constraint, `l2 <: -{poison}` says that `l2` is any label that has not been tainted by the `poison` atom. The second constraint, `l1 <: (l2 ⊔ +{poison})`, says that it is not possible to lower `l1` in any way other than removing

the taint of the `poison` atom. Otherwise it would be possible to instantiate `scrub` so that it is a function that coerces from `α @ ⊤` to `α @ ⊥`, which is far too general.

It may seem a strange that `scrub` quantifies over a function from labels to types instead of a type. The problem with quantifying directly over types with a base kind, is that `scrub` would wind up with the following type:

```
fun scrub : ∀⟨l1 l2|α: Lab -(+)-> % @ l1|l2 <: -{poison} &
                                        l1 <: (l2 ⊔ +{poison}) &
                                        (info α) = l1 &
                                        (info α) = l2⟩ α -(l2)-> α
```

Because there is no way to directly refer to the information content of α, it is necessary to use `info` labels. However, if `info` labels are used, it clearly becomes impossible to ever call `scrub` with distinct labels for `l1` and `l2` because of transitivity. This is an example of where it would be useful for InforML to provide skeleton constraints like FlowCaml (§ 3·6).

One possible implementation of a scrubber for the `companies` module can be found in Figure 4·10. Its functionality is split into two distinct parts. The functions `correct` and `correctSpine` recursively traverse an input and update embedded `Dept` data constructors so that their cached valuation is the sum of the valuation of the `Dept`'s `manager` and `SubUnits`. The function `scrub` simply calls `correct` on its input, and then uses the primitive function `declassify` to remove the `poison` atom from the information content of the input.

The function `declassify` has the type:

```
∀⟨l1 l2 l3|α: Lab -(+)-> * @ l3|l2 <: l1⟩ α @ l1 -(⊤)-> α @ l2
```

It is the only mechanism in InforML for lowering the information content of a value. Because `declassify` can lower the information content of values, it can be used to bypass restrictions on valid information flows imposed by the type system. Therefore, it amounts to an *unsafe* downcasting mechanism for labels and should only be used judiciously.

There are language extensions that could be used to restrict the scope of `declassify`. On possible extension would be to consider a label lattice that is closer in structure to the Decentralized Label Model (DLM) of Myers and Liskov (2000). For example, if InforML were extended with a notion of ownership or principals, the `companies` module could then be treated as the owner of the abstract data types it defines, and, more importantly, the owner of the atom `poison`. As the owner of the `poison` atom, only code written inside the `companies` module would be allowed to use the `declassify` to remove the `poison` atom from labels. Any code outside of the scope of the module will not be able to use `declassify` to remove the atom. However, with such an extension the onus is still on the author of a module to use the `declassify` function responsibly.

Now that I have defined the break and recover version of the `companies` module, it is now possible to write the type-directed function `increase`, while still maintaining integrity. The implementation of this version of `increase` is given in Figure 4·11. Similar to `scrub`, the implementation of `increase` is broken up into two parts. The first part is just a wrapper function that calls the second part and then uses the `scrub` function to remove the taint from the result. The second part performs the actual type-directed traversal that increases the salary.

```
fun correct : ∀⟨l|α: * @ l|(info α) = l⟩ α -(l)-> α
fun correctSpine : ∀⟨l|α: * @ l|(info α) = l⟩ Spine α @ l -(l)-> α

fun scrub : ∀⟨l1 l2|α: Lab -(+)-> % @ l1|l2 <: -{poison} &
                                          l1 <: (l2 ⊔ +{poison})⟩
            α @ l1 -(l2)-> α @ l2

fun correct ⟨l|α⟩ arg =
    typecase α
           | Dept @ l =>
             case arg of (D ⟨l⟩ nm mn _ sbs) =>
               let
                 val sbs' = list.map (correct ⟨l|(SubUnit @ l)⟩) sbs
               in
                 D nm mn ((valuation' ⟨l|(manager @ l)⟩ mn) +
                          (valuation' ⟨l|(List (SubUnit @ l) @ l)⟩) sbs') sbs'
               end
             end
           | _ => isdata α then
                    correctSpine ⟨l|α⟩ (toSpine ⟨l|α⟩ arg)
                  else
                    arg
                  end
         end

and correctSpine ⟨l|α⟩ spn =
    case spn
       | SHead dc => dc
       | SCons ⟨l1 l2|β ψ⟩ newspn arg =>
           (correctSpine ⟨l2|(β -(⊤|l)-> ψ)⟩ newspn) (correct ⟨l2|β⟩ arg)
       | SConsEx _ _ =>
           abort "Unexpected input data type."
     end

and scrub ⟨l1 l2|α⟩ arg = declassify ⟨l1 l2 l1|α⟩ (correct ⟨l1|(α @ l1)⟩ arg)
```

Figure 4·10: A scrubber for the `companies` module

```
fun increase : ∀⟨l1 l2|l2 <: -{companies.poison} &
                       l1 <: (l2 ⊔ +{companies.poison})⟩
      (| companies.Company @ l2, Int @ l2 |) -(l2)-> companies.Company @ l2

fun increaseInternal : ∀⟨l|α: * @ l|(info α) = l & l :> +{companies.poison}⟩
                  (| α, Int @ l |) -(l)-> α

fun increaseSpine : ∀⟨l|α: * @ l|(info α) = l & l :> +{companies.poison}⟩
                      (| Spine α @ l, Int @ l |) -(l)-> α

fun increase ⟨l1 l2|⟩ arg amt =
    companies.scrub ⟨l1 l2|companies.Company⟩
       (increaseInternal ⟨l1|(companies.Company @ l1)⟩ arg amt)

and increaseInternal ⟨l|α⟩ arg amt =
      typecase α
             | companies.Salary @ l =>
                  companies.newSalary (((companies.valSalary arg) * amt) div 100)
             | _ => isdata α then
                       increaseSpine ⟨l|α⟩ (toSpine ⟨l|α⟩ arg) amt
                     else
                       arg
                     end
          end

and increaseSpine ⟨l|α⟩ spn amt =
    case spn
       | SHead dc => dc
       | SCons ⟨l1 l2|β ψ⟩ newspn arg =>
           (increaseSpine ⟨l2|(β -(⊤|l)-> ψ)⟩ newspn amt)
             (increaseInternal ⟨l|β⟩ arg amt)
       | SConsEx ⟨l1 l2 l3 |β ψ⟩ newspn arg =>
             ifholds l2 <: l3 & (info β) = l3 then
               (increaseSpine ⟨l3|(β -(⊤|l)-> ψ)⟩ newspn amt)
                 (increaseInternal ⟨l|β⟩ arg amt)
             else
               abort "Cannot create a value with the requested information content"
             end
     end
```

Figure 4·11: A break and recover implementation of the `increase` function.

Overall, the two functions that perform the actual work, `increaseInternal` and `increaseSpine`, are very similar to the implementation presented at the beginning of the chapter. However, the three differences are an additional constraint on the quantified label `l :> +{companies.poison}`, a nontrivial case for `SConsEx` in `increaseSpine`, and the use of accessor and constructor functions provided by the module.

The constraint is necessary because when the type pattern `companies.Salary @ _` is matched in `increaseInternal`, the program counter is raised by `+{companies.poison}`. However, the overall type of the function is expecting a value of type α. Because the information content of α is expressed indirectly, it is not possible to express that the information content of α will be raised by `increaseInternal`. Therefore, instead of expressing a change in the information content of α, the constraint `l :> +{companies.poison}` is used to make sure that `l` is instantiated with a high enough information content that the information content of α does not need to change. This requirement does not prevent `increase` from being used on unrestricted data labeled with ⊥. When `increaseInternal` is called by `increase` the label will be raised by subsumption to exactly `+{companies.poison}` and then the label on the resulting value will brought back down to ⊥ by `companies.scrub`.

The reason that `increaseSpine` must try to handle the `SConsEx` case, unlike the other examples I have shown so far, is that `increaseSpine` must be implemented with no knowledge of how the abstract types in the `companies` module are implemented. It is entirely possible that, during the recursive traversal, `toSpine` can encounter a hidden data structure that can only be converted to a spine using `SConsEx`. However, `increaseSpine` and `increaseInternal` can only construct an appropriate value if `l2` and the information content of β match what is needed. Otherwise, `abort` will be called to report a dynamic failure.

The fact that `increaseInternal` can use an accessor function, unlike in the previous section, requires some explanation. The reason it was impossible to use `companies`'s accessor functions while writing `increase` was because of their restricted program counters. Here, because `newSalary` and `valSalary` have been exposed as label polymorphic functions, it is possible to instantiate them with a label that allows them to be called even with a restricted program counter.

The `increase` function itself requires two label arguments and constraints identical to `scrub` – otherwise it would not be possible to call `scrub`. However, it calls `increaseInternal` with the tainted label, in order to satisfy its constraints.

It is now possible to formally state the integrity property for the break and recover idiom.

Conjecture 4·4·1 (Integrity for break and recover)**.** *Assuming that the program only contains legitimate uses of the* `declassify` *function, for abstract data types labeled with some atom* `a`*, any function that requires*

- *its inputs have an information-content less than* `-{a}`*,*
- *that it may only be called in contexts where the program counter label is less than* `-{a}`*,*

can assume that all invariants preserved by the implementation of the abstract data type will hold.

For example, `increase`, is guaranteed that any value it produces will not violate the integrity of the `valuation` function. Namely, any values of type `Dept` that `valuation` receives will have a valid cached valuation.

5

The design and implementation of the InforML language

> Everyone by now presumably knows about the danger of premature optimization. I think we should be just as worried about premature design - designing too early what a program should do.
>
> Paul Graham (*Hackers and Painters*, 2003)

The development of the InforML language has been a long process and I have learned many things along the way. Some of what I have learned is about trade-offs in the implementation of programming languages and compilers that are not addressed in texts on language implementation, if even in research papers. However, most of these things are not relevant to InforML specifically, so I will not cover them in this dissertation. Some of what I have learned is about trade-offs and issues in the design of a language that features type-directed programming with an information-flow type and kind system. These trade-offs will be the primary focus of this chapter. As with most things learned from practical experience, the lessons were that I had not made the best choice when approaching these trade-offs. I have already explained some of these choices in passing in § 3 and § 4.

I will begin the chapter with a brief introduction to the implementation of the InforML language. I will then follow with discussions of what I believe to be the design choices that have the greatest influence on the character of the language: merging type constructors and types, using the `toSpine` primitive for analyzing generative data types, and including existential labels in the language. I will then conclude the chapter with a discussion of some less significant design choices, that were nevertheless not obvious, but important to discuss for posterity.

§ 5·1 The implementation of InforML

It is important to view the implementation of InforML that I have built as a tool for answering questions about TDP with an information-flow type and kind system, not as a tool that will ever be used to write software.

The implementation of InforML is a menagerie of different programming language technologies. The majority of InforML was implemented in Standard ML (Milner et al. 1997), with some custom extensions to SML/NJ to provide syntactic sugar for monads. The InforML parser was written using Turon's (2007) new LL(k) parser generator, and its accompanying Unicode (Consortium 2006) friendly lexer. Some of the implementation is written using noweb (Ramsey 1999) in an attempt to provide more detailed documentation.[1] The m4 macro processor (Kernighan and Ritchie 1977) was used in several cases to work around limitations of the SML/NJ compilation manager (Blume 2007). Some parts of type inference are handling by passing on logic programming queries to the Twelf logical framework (Pfenning and Schürmann 1999). Finally, the InforML runtime is written in Scheme (Sperber, Dybvig, Flatt, and van Straaten 2007).

The InforML implementation is roughly divided into three stages: the frontend, the typechecker, and the compiler. Unfortunately, at this time the compiler stage is no longer functioning, as it has not been possible to keep pace with changes to typechecker that have resulted from writing this dissertation.

¶ **The frontend** The frontend is relatively uninteresting, aside from the issues surrounding LL(k) parsing. Using a LL(k) parser versus the more common choice of a LALR(1) did impact the design InforML, but this was mostly confined to the syntax. For example, anonymous functions, conditionals, case statements, etc. all are closed with the `end` keyword to make the LL(k) grammar simpler. On the other hand, because it is possible to look ahead more than a single token, it is possible to make use of the Haskell-style `fun x : σ` type signatures for functions. After parsing, the frontend processes the entire syntax tree, eliminating syntactic sugar and some variable renaming, to produce an abstract syntax tree in the format used by the typechecker.

¶ **The typechecker** The typechecking stage is the largest in terms of source code and is also the most complex stage. This stage also had the most significant impact on the design of InforML. A significant amount of time was spent on developing an implementation of global type inference for InforML. However, this implementation was extremely buggy and was eventually scaled back to local type inference, because the type inference problems InforML present are orthogonal to the thesis of this dissertation.

In retrospect, I think most of the difficulty in developing a correct implementation of global type inference stemmed from attempting to incrementally resolve constraints while traversing the abstract syntax tree. I think that the complexity would have been much more manageable had I cleanly separated constraint generation from constraint solving. Another problem was, that for a few months, I attempted to use de Bruijn indices with explicit substitutions (Abadi, Cardelli, Curien, and Lèvy 1990) in an attempt to simplify some of the issues of binding and scope during unification. This proved to be a terrible

1. However, like most source code documentation, it proved difficult to keep up to date with the many radical changes made to the InforML internals.

mistake. They introduced their own problems, greatly expanded the printed size of types, labels, and kinds, and made reading debugging output a slow and painful task. Some blame for the failure to build a working implementation of global type inference can probably also be attributed to typical graduate student over-ambition.

Using local type inference for InforML, rather than global type inference, has a significant impact on the feel of the language. For example, assuming a global inference algorithm has been implemented, a programmer would have to expend less effort in crafting "optimal" type signatures and data type definitions. For example, when writing mutually recursive functions, I would frequently begin by writing the functions without any constraints on their label and type arguments, and then manually attempt to reach a fixed point on the best constraints by repeated interaction with the typechecker. This was usually a fairly painful process of chasing labels around programs.

Despite the limitations of local type inference, implementing global type inference for InforML is not a trivial undertaking. I conjecture that a global type inference algorithm for InforML approaches the difficulty of theorem proving for at least Π^0_2 first-order logic, simply to solve constraints on the label lattice. If the constraints that the user can write are carefully restricted, it may be possible to eliminate the disjunctive (but not existential) non-determinism in the formulas generated (Pottier and Simonet 2003). Because subtyping is defined in terms of lifting label subsumption to types, I conjecture that all type constraints will be equational and could be restricted so that they can be decided using higher-order pattern unification (Miller 1991).

Probably the most interesting aspect of the typechecking stage is that subkinding, subtyping, and constraint checking were all initially implemented by encoding the problems as logic programming queries handed off to the Twelf logical framework (Pfenning and Schürmann 1999). The primary motivation for this was that nearly all of the InforML type system could be elegantly specified as a logic program in the LF meta-logic (Harper, Honsell, and Plotkin 1993). Not only was it easy to cleanly specify InforML, but I was able to implement most of the key parts of the type system in an afternoon. Only a few more days were required to completely describe the language. Furthermore, representing the InforML language using higher-order abstract syntax (Pfenning and Elliott 1988) made the correct static semantics for some parts of the language, like the use of internal versus external names in modules (Harper and Lillibridge 1994), "fall out" of the specification naturally.

Despite these benefits, using Twelf in this fashion had a number of significant disadvantages:

- Because I specified the InforML type system in a relatively declarative fashion, it is very easy for Twelf to "diverge" on incorrect programs while it searches for a witnessing proof.
- Even for correct programs, Twelf can become lost while searching for a proof.
- If Twelf does determine that a query has no solutions, there is no easy way to translate this result back into a reasonable error message.

- Because Twelf is a research project itself, some of the experimental features I tried to use to improve proof search would cause exceptions in Twelf, or in one case return wrong answers.[2]

The second and third issues I was able to resolve to some extent by reimplementing many of the common queries that did not require label unification variables within SML. I was also able to address the second issue some by rewriting my specification of the InforML type system in a more algorithmic style. Switching to a sequent style formulation (Gentzen 1935) for the subsumption and constraint checking rules proved to help significantly.

However, it is not possible to completely eliminate the use of Twelf without putting label unification variables back into the parts of InforML implemented in SML. Specifically, the usual rule polymorphic subsumption requires guessing a label:

$$\frac{[\Pi/l]\sigma_1 <: \sigma_2}{\forall\langle l\rangle\ \sigma_1 <: \sigma_2}$$

The rule is usually implemented by substituting a fresh unification variable for l. Unlike inferring label and type instantiations for polymorphic functions, I cannot use a simplistic and incomplete matching heuristic. If InforML cannot infer label and type instantiations for a polymorphic function using this heuristic, the programmer can just supply the instantiations herself. It is not as reasonable, in my opinion, to make the subtype checking algorithm incomplete and ask the programmer to supply a subtyping proof when it fails.

¶ Compilation Finally, the compilation stage takes a well-formed InforML program and generates Scheme code from it. In terms of language targets, I think Scheme is an excellent choice. Ideally, I would target a statically typed language so that it generated code can be verified statically. However, when implementing experimental languages it is often the case that there does not exist a language that will allow a naïve encoding of your language to typecheck. For example, I could not have compiled InforML to the ML family of languages because there would be no way straightforward way to encode polymorphic recursion.[3] In retrospect, I conjecture that with some additional time it may have been possible to target the GHC Haskell compiler (Peyton Jones, Hall, Hammond, Partain, and Wadler 1993), but it would have probably taken significantly longer. On the other-hand, Scheme is a much better choice than other popular high level language targets like C, because it provides more suitable abstractions. Furthermore, there are a several decent Scheme implementations to choose from.

Because labels and types are an important part of InforML's operational semantics, labels and types are compiled to Scheme values representing them. I have represented labels using tagged sets of symbols, where the tag specifies whether the set is additive or subtractive. Following past work on compiling languages with runtime type analysis (Crary, Weirich, and Morrisett 2002), types are compiled to representations in a fashion very similar to data constructors; type functions are in fact compiled to

2. Specifically, I found a logic program query that the tabled logic programming engine would report as having no solutions (which I believe to be the correct answer) while the theorem prover would report a solution (but not provide a witnessing proof).

3. There is a workaround for this in recent versions of OCaml by making using of its support for recursive modules and functors.

term level functions. Compilation converts type and label arguments of term functions into additional term arguments.

Term and type pattern matching compilation takes advantage of first-class continuations to generate simpler code for handling match failures. Term and type patterns are both decomposed into sets of boolean preconditions and projections that extract components of their input and bind them to variables.

Despite what I discussed in § 3·3, data constructors are actually compiled to functions that return lists starting with a symbol naming the data constructor, followed by a representation of the data constructor's type, its label arguments, its type arguments, and its value arguments. Compiling data constructors as functions simplified compilation, because the alternate approach would have required an analysis to distinguish whether a term application is a function application or a data constructor application. Additionally, because InforML allows data constructors to be used as curried functions, it would have been necessary to insert η-expansions regardless.

The reason that I compile data constructors with an embedded copy of their type representation is so that `toSpine` can examine the type representations to decide whether a given argument should be placed in a `SCons` or and `SConsEx` node. Unfortunately, this opens a loop-hole for breaking confidentiality.

Consider the following module:

```
module m : sig
  type t : Lab -(+)-> * @ ⊤
  data F : : Lab -(+)-> % @ ⊥
  cons MkF : ∀⟨l⟩ t @ l -(⊤)-> F @ l
  val x : t @ ⊥
end = mod
  type t = Int

   datatype F : Lab -(+)-> % @ ⊥ =
     | MkF : ∀⟨l⟩ t @ l -(⊤)-> F @ l

   val x = 3
end
```

When the data constructor `MkF` is compiled it will be tagged with the type `Int @ l -(⊤)-> F @ l`, where `Int @ l` has kind `* @ ⊥`, and `l` is bound by an enclosing Scheme `lambda`. Therefore, the following code written outside module `m` will behave in an unexpected fashion:

```
case toSpine ⟨⊥|(m.F @ ⊥)⟩ (m.MkF ⟨⊥⟩ m.x)
   | SCons ⟨l1 l2|α β⟩ _ _ => True
   | SConsEx _ _ => False
 end
```

Because `m.MkF` was compiled with a type representation that indicated its argument has a type with an information content of ⊥, and because the signature I ascribed to `m` is erased by the compiler, when `toSpine` is applied to `(m.MkF ⟨⊥⟩ m.x)` in the code above, `toSpine` will inspect this stored representation and conclude that its first argument has a kind with an information content of ⊥, and it can therefore safely construct a `Spine` using a `SCons` node for its argument `m.x`. Furthermore, inside the `case` branch for `SCons`, the label variables `l1` and `l2` will be bound to ⊥ at runtime, and the type variable β to `Int @ ⊥`.

Except that β is just another name for `m.t @ ⊥`. Changing `m.t @ ⊥`, which has kind `* @ ⊤` will therefore change the observable type bound to β, which has the kind `* @ ⊥`. Therefore, a low security observer can now distinguish between changes to a higher security type – confidentiality has been broken. However, integrity still holds, because there is no way to relate values with type β back to values of type `m.t @ ⊥` without them becoming tainted by ⊤.

The loop-hole could be closed by giving module signatures an operational meaning, inserting a coercion at compile-time that will rewrite the labels inside a module to be consistent with the signature. In other words, a coercion semantics for module subsumption (Breazu-Tannen, Coquand, Gunter, and Scedrov 1991). However, properly designing such a coercion semantics will require further study. The other alternative, would be to eliminate `toSpine` from the language in favor of another solution. I will discuss some of the other problems `toSpine` presents in § 5·3.

Otherwise, so far, the implementation of the compilation stage has had little impact on the design of InforML. The necessity of having a working implementation of `toSpine` written in Scheme, as part of the InforML runtime, helped clarify its semantics, specifically with regards to the need for having both the `SCons` and `SConsEx` data constructors.

§ 5·2 Merging type constructors and types

In retrospect, I do not think I would have chosen to use a combined language of types and type constructors in InforML. However, I do not believe I would have reached this conclusion before I began implementing InforML and writing larger examples in it.

As I discussed in § 3·1, because InforML does not have injections that are explicitly labeled with their information content, it has the convention that the information content of a value is taken from the last label application in a type, `info (τ @ Л) = Л`. This had two consequences: the need for `info` labels, and a restriction on label functions. This restriction is that for a function from labels to types, `λ l:Lab =(π)=> τ end`, that the equality `(info τ) = l` hold. This restriction is to prevent these sorts of type functions from being used to discard the information content of a value by writing an abstract type like `λ l:lab =(+)=> Int @ ⊥ end`. By keeping types and type constructors separate, there would be no longer be a need for `info` labels, and subsequently no need to have the restriction on functions from label to types (or rather type constructors).

Furthermore, as I demonstrated with my example in § 3·1, quantifying over types at kind `Lab -(+)-> * @ Л` cannot be used to simulate an explicit injection from type constructors to types. This is because there exist types, such as tuples, with a normal form that cannot be given this kind.

The `info` labels in InforML are not strictly a problem directly, but have the unintended consequence of making the language of kinds, types and labels mutually recursive. Such mutual recursion complicates the implementation of InforML and is sure to complicate the metatheory of InforML. Additionally, when writing most polymorphic functions, I have found that is necessary to include a constraint of the form `(info α) = Л`, which is just pushing the label that would be always be available on an injection in $\lambda_{\text{SEC}i}$ into a constraint.

Having explicitly labeled injections from type constructors to types also eliminates the motivation for potentially extending InforML with skeleton constraints, as I discussed in § 3·6 with respect to FlowCaml.

The addition of skeleton constraints would resolve some of the shortcomings of `info` labels, but I think that the better resolution would be to make the language simpler by eliminating `info` labels, rather than solving the problem by making the language more complex with an additional form of constraint.

§ 5·3 The `toSpine` primitive

As I discussed in § 3·3, the primitive function `toSpine` is included in InforML to make it possible to write type-directed operations over algebraic data types. However, `toSpine` is unsatisfactory in a number of ways. First, the implementation of the `toSpine` combines two orthogonal language capabilities. Second, the `Spine` data type itself has limitations and problems. Finally, some type-directed operations that are desirable cannot be written for algebraic data types using `toSpine`.

Based upon my explanation of `toSpine` § 3·3, it is clear that the operational semantics of `toSpine` are nontrivial. As such, I think it would be better to instead provide its capabilities in the form of two or more orthogonal language features. For example, in order to decide whether it should build a `Spine` value using the `SCons` data constructor or the `SConsEx` data constructor, `toSpine` must internally perform label analysis on data constructors. This functionality is already present with the dynamic constraint checking primitive `ifholds`.

Internally, `toSpine` also is able to pull out the arguments of arbitrary data constructors, because it can take advantage of the fact that all data constructors have a uniform representation in memory. It seems sensible to somehow make this capability accessible in a more direct fashion. There are a multitude of approaches that have been taken (Lämmel and Peyton Jones 2003, 2004, 2005; Hinze, Löh, and Oliveira 2006; Hinze and Löh 2006; Weirich 2006; Mitchell and Runciman 2007), but further study is necessary to determine which is most appropriate for use in an information-flow type and kind system . As I will discuss in § 6·1, I think the correct step is to try to understand how `toSpine` really works, by looking at a language with a more primitive notion of type generativity.

The `Spine` data type by itself is also problematic, something that I was aware of from the outset. In InforML, it is only possible to construct `Spine`s from algebraic data types of base kind (`% @ Л`). However, there are many type-directed operations that can only be defined over type functions. For example, there are many interesting type-directed operations on "container" types: maps, folds, etc. Such functions would take a type function with the kind of the form $(* \mathbin{@} Л_1) \text{ -(}\pi\text{)-> Lab -(+)-> } (* \mathbin{@} Л_2)$ as an input. In fact, the `Spine` data type is merely the base case in an infinite hierarchy of `Spine`-like structures parametrized by types of different kinds. It might be possible to resolve this issue by extending InforML with kind polymorphism and what are called *polykinded types* (Hinze 2000). A polykinded type has a structure that is inductively defined for the structure of a kind.

Also related to the structure of the `Spine` data type, is that because `Spine`s are intended as uniform "views" of a data constructors, it is only possible to implement type-directed operations over existing instances of an algebraic data type. For example, I gave an implementation of `toString` in § 3·5 that used `toSpine` to handle the recursion over algebraic data types. But there is no way to write the dual function, `fromString`, in InforML. This is because there is no way to create a data constructor from just its name as a string. This calls for a completely new primitive function or language feature. Hinze and

Löh (2006) have explored how to address this limitation of `Spines` by putting more information into the type representations they use in their implementation, but there may be other more pleasing solutions.

Finally, another problem with the definition of the `Spine` algebraic data type is that the `SConEx` data constructor contains an existentially quantified label. As I will describe in the next section, existentially quantified labels are difficult to program with and can prevent precise reasoning about confidentiality and integrity. However, I conjecture that any alternative to the use of `Spines` will introduce existentially quantified labels in some form.

§ 5·4 Existential labels vs label analysis

Early in the development of InforML, allowing existentially quantified labels seemed like a sensible idea. Existential labels seemed to offer the ability to gracefully degrade from static enforcement of information flows to dynamic enforcement of information flows. Combined with dynamic constraint checking, it would even be possible to switch back from dynamically tracking information-flows to statically tracking them. Finally, as I mentioned in the previous section, because the plan was to use `toSpine` to allow TDP with algebraic data types, existentially quantified labels were also necessary to give the `Spine` data type's data constructors satisfactory type signatures. However, in retrospect, I think that in a revised version of InforML it would be best to attempt to minimize the use of existential labels, if not eliminate them altogether.

The first problem with existentially quantified labels is that it is simply difficult to use them effectively while maintaining the hidden labels with any precision. The second problem is that existential labels weaken the claims that can be made about confidentiality.

I will illustrate the difficulties with programming with existential labels, by revisiting the dynamic type (Abadi, Cardelli, Pierce, and Plotkin 1991), `Dyn`,that I introduced in § 3·3:

```
datatype Dyn : Lab -(+)-> % @ ⊥ =
  | Dynamic : ∀⟨ld l|α : * @ ld⟩ α -(⊤)-> Dyn @ l
```

Now here is function that tries to implement addition on dynamic values (that are integers):

```
fun addDyn : ∀⟨l⟩ (| Dyn @ l, Dyn @ l |) -(l)-> Option (Dyn @ l) @ l
fun addDyn ⟨l⟩ d1 d2 =
    case (d1, d2)
       | (Dynamic ⟨l1 l|α⟩ x, Dynamic ⟨l2 l|β⟩ y) =>
         typecase (α, β)
                | (Int @ l3, Int @ l4) =>
                  Some (Dynamic ⟨(l ⊔ l1 ⊔ l2 ⊔ l3 ⊔ l4)
                                (l ⊔ l1 ⊔ l2 ⊔ l3 ⊔ l4)|
                                (Int @ (l ⊔ l1 ⊔ l2 ⊔ l3 ⊔ l4)))
                                (x + (⟨l ⊔ l1 ⊔ l2 ⊔ l3 ⊔ l4⟩ y))
                | _ => None
              end
     end
```

The above function will fail to typecheck because the body produces a value with the type

```
Option (Dyn @ (l ⊔ l1 ⊔ l2 ⊔ l3 ⊔ l4)) @ (l ⊔ l1 ⊔ l2 ⊔ l3 ⊔ l4)
```

and there is no guarantee that `l ⊔ l1 ⊔ l2 ⊔ l3 ⊔ l4` will be less than `l`. It is not an option to change the definition of `Dyn` and `Dynamic`,[4] but I can make `addDyn` typecheck by making use of `ifholds` :

```
fun addDyn : ∀⟨l⟩ (| Dyn @ l, Dyn @ l |) -(l)-> Option (Dyn @ l) @ l
fun addDyn ⟨l⟩ d1 d2 =
    case (d1, d2)
       | (Dynamic ⟨l1 l|α⟩ x, Dynamic ⟨l2 l|β⟩ y) =>
         typecase (α, β)
                | (Int @ l3, Int @ l4) =>
                  ifholds (l1 ⊔ l2 ⊔ l3 ⊔ l4) <: l then
                    Some (Dynamic ⟨l l|(Int @ l)⟩ (x + y))
                  else
                    None
                  end
                | _ => None
             end
     end
```

However, this function is nearly useless. It requires the caller to guess a label to instantiate `addDyn` with that she hopes will make it correctly add the two dynamic values (assuming they are both integers). The only way to guarantee that the function will add the inputs (when they are integers) is to instantiate the quantified label with ⊤. And at that point, I could have just written the function to return a value with an information content of ⊤ in the first place:

```
fun addDyn : (| Dyn @ ⊤, Dyn @ ⊤ |) -(⊤)-> Option (Dyn @ ⊤) @ ⊤
fun addDyn d1 d2 =
    case (d1, d2)
       | (Dynamic ⟨l1 l2|α⟩ x, Dynamic ⟨l3 l4|β⟩ y) =>
         typecase (α, β)
                | (Int @ _, Int @ _) =>
                  Some (Dynamic ⟨⊤ ⊤|(Int @ ⊤)⟩ (x + ⟨⊤⟩ y))
                | _ => None
             end
     end
```

This version of `addDyn` trades precision in tracing information flows for reliability – the caller is guaranteed that `addDyn` will not silently fail (or in an alternate implementation abort execution).

The problems existential labels pose for reasoning about confidentiality are similar: functions must either use more conservative labels, making them appear to violate confidentiality of more abstractions than they truly do, or functions must be made partial. This trade-off has shown up in nearly all of the type-directed functions written in InforML that are intended to consume arbitrary data as an input.

For example, the version of `toString` in § 3·5 is "partial" in the sense that for some inputs it may return a string containing `<Redacted>` to indicate that it encountered data that it could not process and still return data with the requested label. Alternately, the implementation of `increase` given in § 4·2 will simply abort execution if it finds that it must process data with a higher information content than it can handle and still meet its type specification.

4. It is an option for me, but that is only because I wrote the InforML basis library.

The alternative, to give these functions more conservative labels, is not one I have chosen to use. This is because the most precise, yet conservative, labels that I could have given to the outputs of `toString` and `increase` would be `⊤`. This highly constrains the use of the values that type-directed functions produce, because the authors of most modules, when using the harmless reflection idiom and the break and recover idiom, for example, will use type signatures to prevent the use of tainted data as inputs. Therefore, I have usually favored making functions partial, because runtime failures should be rare enough to make it the better trade-off.

Neither of these alternatives are satisfactory. With partiality, a user of a function may not be able to predict the runtime behavior of type-directed function, as they may have no way of knowing that hidden inside their input is an existential label. Alternately, conservative labeling make its difficult for the user of a function to reason about which flows will actually occur, they can only assume that any possible flow may arise.

Therefore, because of the difficulty of writing programs that make use of existential labels to dynamically trace information flows, and the fact that existential labels make reasoning statically about confidentiality difficult, in the future it would be best to either examine techniques for minimizing the use of existential labels or techniques for making existential labels easier to work and reason with.

§ 5·5 Other design trade-offs

In this section I will briefly discuss some other design trade-offs and choices that I do not think significantly impact the nature of InforML, but are nonetheless worth noting for posterity.

§ Defining algebraic data types

In InforML, the programmer defines an algebraic data types with a syntax that is similar to what was chosen for defining GADTs (Coquand 1992; Crary and Weirich 1999; Xi, Chen, and Chen 2003; Peyton Jones, Vytiniotis, Weirich, and Washburn 2006) in Haskell. However, it may be more sensible to treat data constructors like OCaml does, and require data constructors to be fully applied at their use site. InforML, Standard ML, and Haskell all allow data constructors to be used as if they were functional values.

If data constructors are not defined in a fashion that makes them appear to have functional types, it is no longer necessary to provide the vestigial program counter and function closure labels that show up in InforML. However, defining data constructors in terms of a functional type does perhaps make defining GADTs more intuitive, but there may exist syntactic sugar for that purpose.

§ Subsumption and pattern matching

While implementing InforML I discovered that for typechecking `case` and `typecase` expressions it was possible to choose between two possible semantics. For example, consider the following program fragment:

```
case x : Int @ l1 of y : Int @ l2 => ... end
```

What should the relationship between `l1` and `l2` be? Strangely, the language design could arguably choose between two options.

- `l1` must be less than or equal to `l2`. This is a reasonable answer because it matches the standard substitution lemma: if `e` has type `τ1`, the variable `x` has type `t2`, and `τ1 <: τ2` then it is sound to substitute `e` for `x`.
- `l2` must be less than or equal to `l1`. This is a reasonable answer, but is only sound if `case` expressions perform label analysis. By label analysis, I mean that control flow of the pattern match will be affected by what `l1` is at runtime. For example, in the following code fragment the first branch will never execute unless `l` is equal to $\bot$ at runtime:

```
case x : Int @ l
  of y : Int @ ⊥ => ...
   | y : Int @ ⊤ => ...
 end
```

For InforML, I chose to use the first semantics, because it does not require term and type pattern matching to also perform label analysis at runtime.

The choice between these options for term level pattern matching does not have much of an effect on the expressive power of InforML. This is because there is already `ifholds` that can be used to analyze labels at runtime. Therefore, it is undesirable because it duplicates existing functionality and complicates the implementation because `case` would need to be able to dispatch on labels, like `ifholds`, as well as values.

However, choosing the latter option for type level pattern matching would alter the expressive power of InforML. Because kinds are presently erased during compilation, there is no mechanism for getting at their labels at runtime. Therefore, it would be necessary to compile and pass around kind representations at runtime. Furthermore, it does not seem sensible to combine this orthogonal functionality into the `typecase` operator.

§ The information content of tuples

In § 3·1 I explained that the equivalences for the `info` label on tuples was the following:

$$\texttt{info}\ (\tau_1 \ldots, \tau_n) \quad = \quad \texttt{info}\ \tau_1 \ = \ \cdots \ = \ \texttt{info}\ \tau_n$$

This definition is an extremely recent change to the language definition. Previously I used the definition:

$$\texttt{info}\ (\tau_1 \ldots, \tau_n) \quad = \quad \texttt{info}\ \tau_1 \ \sqcup \ \cdots \ \sqcup \ \texttt{info}\ \tau_n$$

This definition allows for a little more flexibility in constructing tuples, because each component is allowed to have an independent information content.

However, this latter definition for the information content of a tuple makes it nearly impossible to write some type-directed functions on tuples. For example, consider my initial example from § 3·1, `toString`:

```
fun toString : ∀⟨l:Lab|α: * @ l|(info α) = l⟩ α -(l|⊥)-> String @ l
fun toString ⟨l|α⟩ arg =
    typecase α
           | Bool @ l      =>
             if arg then "True" else "False" end
           | _ -(_ | _)-> _ =>
             "<Function>"
           | (β, ψ)         =>
             "(" ^ (toString ⟨l|β⟩ (arg.0)) ^ "," ^ (toString ⟨l|ψ⟩ (arg.1)) ^ ")"
         end
```

If the branch for tuples were to be typechecked using the old definition of the information content of tuples, it will quickly fail.

```
...
           | (β, γ)      =>
             "(" ^ (toString ⟨l|β⟩ (arg.0)) ^ "," ^ (toString ⟨l|γ⟩ (arg.1)) ^ ")"
...
```

In order to call `toString` recursively on the first and second projections of `arg`, it must be the case that `(info β) = l` and `(info γ) = l`. In this particular branch of the `typecase`, it is known that `(info α) = l` and that `α = (β, γ)`. Substituting for `α` and using the definition of `info`, I can conclude that `(info β) ⊔ (info γ) = l`. However, this is not enough to show that `(info β) = l` and `(info γ) = l`.

I can decompose `(info β) ⊔ (info γ) = l` into the constraints `(info β) ⊔ (info γ) <: l` and `(info β) ⊔ (info γ) :> l`. The first constraint implies that `(info β) <: l` *and* `(info γ) <: l`, which is half of what is needed. However, the second constraint implies that `(info β) :> l` *or* `(info γ) :> l`. There is no guarantee that both `(info β) :> l` and `(info γ) :> l` hold, just that one of them must. Therefore, with the old definition of `info` for tuples, it is not possible to typecheck this version of `toString`.

It is plausible that the problem is that the precondition I have chosen for `toString` in this case is simply too strong. However, if I relax it to `(info α) <: l`, the sub-expression `arg.0` will be ill-typed because the program counter label is `l`, yet the information content of `arg.0` must be greater than or equal to `l`. However, the information content of `arg.0` is `info β`, which is only known to be less than or equal to `l`, not greater than or equal. Alternately, if I try making the precondition `(info α) :> l`, I encounter the same situation as when the constraint is an equality – that I am only guaranteed that `(info β) :> l` *or* `(info γ) :> l` hold, not both.

Therefore, to resolve this situation, changing the definition of `info` for tuples to the present one seemed the best resolution.

§ 5·6 Conclusion

Probably the most important lesson that I have learned as part of designing the InforML language is the value of having an established metatheory. Without having worked on a metatheory of InforML, it is difficult to guess whether I would have encountered quite as many unexpected surprises while

implementing InforML. However, the time spent working out the theory of InforML may not have revealed the impact some of the design trade-offs have on writing realistic programs. In the next chapter, before concluding this dissertation, I will spend some time discussing my thoughts on future directions for the theories behind InforML.

6

Future work and conclusions

> I don't have to write about the future. For most people, the present is enough like the future to be pretty scary.
>
> William Gibson (2003)

The contributions that I have discussed in the proceeding chapters are valuable, yet there are still many improvements to be considered and new avenues for research to be explored before the ideas I have been presented will be ready for mainstream programming languages. In the next section, I will examine many of the directions for future research, before reviewing my conclusions in the final section.

§ 6·1 Future work

While there is a considerable amount of engineering work to be done on InforML, or some other successor, before the ideas I have described in this dissertation will be ready for use as a mainstream language, I am confident that the engineering issues will be straightforward to solve once the theoretical problems have been addressed. Therefore, most of the directions for future work that I will cover are of a more theoretical nature.

§ A meta-theory for InforML

I feel that InforML's lack of well specified static and dynamic semantics is a significant problem. The implementation of InforML serves partly as an executable specification, but practical concerns make it far removed from a rigorous formal presentation. I believe recent research (Aydemir et al. 2005; Lee et al. 2007; Aydemir et al. 2008) has also begun to show that "paper" formalizations of the meta-theory of programming languages will soon be superseded by *mechanized meta-theory*. I believe that

a mechanical formalization is the correct direction for a language like InforML, especially given the language's complexity.

Once there is a mechanically formalized specification of InforML, the next step will be to prove that the language is type-safe. I have tried my best through testing and debugging the implementation of InforML and my long experience in the design of statically typed languages, but for a language and implementation of the complexity of InforML, there are no doubt still lingering loopholes that allow type safety to be violated. I expect that aside from the challenges introduced by mechanical formalization, the overall proof of type-safety for InforML is unlikely to require the development of new proof techniques, with one exception.

While implementing InforML I have gone back and forth several times on the question of whether subtyping and subkinding in InforML are required to be related in any way. Specifically, if $\tau_1 <: \tau_2$ where type τ_1 has kind κ_1 and type τ_2 has kind κ_2, must it also be true that $\kappa_1 <: \kappa_2$? Currently, I err on not requiring that $\kappa_1 <: \kappa_2$ hold if $\tau_1 <: \tau_2$ in InforML.

The reason for this problem is that to my knowledge, InforML is the first language developed to allow variant dependent kinds. For example, many algebraic data types I have used in my examples have a kind similar to the following:

```
Π l : Lab -(+)-> (* @ l) -(+)-> Lab -(+)-> (% @ l),
```

Here, the first label supplied to the algebraic data type is allowed to vary covariantly during subtyping. So for example, for the `Option` data type in InforML, which has the above kind, `Option @ ⊥`, is a subtype of `Option @ ⊤`. However, something very unexpected happens here: `Option @ ⊥` has kind

```
(* @ ⊥) -(+)-> Lab -(+) -> (% @ ⊥)
```

and `Option @ ⊤` has kind

```
(* @ ⊤) -(+)-> Lab -(+) -> (% @ ⊤),
```

but the former kind is not a subkind of the latter. That is, it is not the case that

```
((* @ ⊥) -(+)-> Lab -(+) -> (% @ ⊥)) <: ((* @ ⊤) -(+)-> Lab -(+) -> (% @ ⊤)).
```

This relationship does not hold because, by the usual subsumption conventions for functional structures, it must be the case that the domains vary contravariantly, `* @ ⊤ <: * @ ⊥`, and the ranges must vary covariantly, `Lab -(+) -> % @ ⊥ <:  Lab -(+) -> % @ ⊤`. The former clearly does not hold. In fact, `Option @ ⊥` and `Option @ ⊤` have completely incomparable kinds.

My current intuition is that this is not a problem. I base this on the strict separation of types and type constructors in $\lambda_{\text{SEC}i}$. In $\lambda_{\text{SEC}i}$, the information content of type constructors can be ignored as soon as they are injected into the language of types. Furthermore, subkinding is only relevant for type constructor well-formedness, while subtyping is only relevant for term well-formedness. However, to prove type soundness, it will be necessary to verify this formally.

Surprisingly, whether the subsumption relationship between kinds must be preserved by subtyping has not been studied anywhere in the literature. To date, research into subtyping with dependent types (Zwanenburg 1999; Aspinall and Compagnoni 2001; Chen 2003) has required that arguments to

dependent functions be invariant under subsumption. Perhaps the closest relevant work is by Martin Steffen (1998) where he studied $\mathbb{F}^{\omega}_{\leq}$ with polarized type applications. However, because his kinds were only first-order it is not immediately obvious how to extended his work to dependent kinds.

§ Generalized parametricity for InforML

Type-safety is unfortunately not enough to be assured of InforML's correctness. Languages with information-flow type (and kind) systems have the unfortunate property that while they can be shown to be type-safe, they may still allow unexpected implicit flows in well-typed programs. That is, the type system can ensure that a well-typed program does not "go wrong" or become stuck, but it is possible that well-typed programs can still leak information. Therefore, it will be necessary to prove generalized parametricity for InforML, or a simplified, but representative, core calculus.

Unlike type safety, I believe that proving generalized parametricity for InforML, regardless of mechanization, will require the development of some new proof techniques. The necessity for new techniques is not specific to the features of InforML. At present there is no entirely syntactic technique for proving standard parametricity, and most of the difficulty in denotational proof techniques used to date have difficulty with realistic programming language features like recursive types and mutable references.

Much like syntactic techniques have proven to scale better for proving type safety (Wright and Felleisen 1994), I conjecture that syntactic alternatives to logical relations proofs are likely to scale better to realistic languages. While there has been some progress in syntactic logical relations proofs (Schürmann and Sarnat), a syntactic proof of parametricity remains an open problem. This is partly a consequence of the considerable expressive power of second-order logic.

I conjecture that it may be possible to extend the proof technique first developed by Pottier and Conchon (2000), and later used by Pottier and Simonet (2003) for proving noninterference for the FlowCaml language, so that a generalized parametricity can be proven syntactically. Their technique for noninterference proofs involves introducing a specialized notion of pairs into the language and showing that the subject-reduction property implies that there is no observable difference between the execution of high-security pairs. Because this is an entirely syntactic technique proof technique, it extends gracefully to languages with mutable state, recursive types and exceptions.

My initial investigations into extending Pottier and Simonet's proof technique have led me to the idea of extending the language with a special "paired" type that corresponds to their paired terms. The critical extension is that these paired types would be labeled with a relation between values of the two types. The remainder of the proof is quite straightforward. However, for a completely formal proof there must be some means of describing the language that defines relations between values, what Schürmann and Sarnat call an assertion logic. Furthermore, it is necessary to show that this logic is sound. However, to be as expressive as the parametricity theorem, the assertion logic must be at least as powerful as second-order logic. Showing that a language of relations based upon second-order logic is sound is equivalent to proving the parametricity theorem. So, in truth, this approach only pushes the difficulty into a different part of the framework.

Since my original investigation of the problem, I conjecture that it may still be possible to prove valuable theorems using a weaker assertion logic. For example, proving the confidentiality and integrity

corollaries in § 2·2 and § 2·2 only requires the universal relation and the empty relation, respectively. Therefore, further research in this direction is warranted.

Should this research direction still fail, there is still a wealth of ideas in this area, so other approaches may be viable. I will briefly discuss some promising starting points.

Pitts has developed a purely operational account of logical relations based upon *biorthogonality* (also called "top-top closure"), but there is no obvious way to extend his methodology to general recursive types (2005). Johann has developed an extension of Pitts's proof to the restricted case where uses of the recursive type must be restricted to be in a positive position (2002). Pitts and Stark have developed an extension that can handle mutable integer references (1998).

Birkedal and Harper have been able to develop a logical relations proof for languages with a single iso-recursive type using what they call *syntactic minimal invariance* (1999). Their formalism is quite involved, but it seems possible that their technique could be applied to problem of extending logical relations proofs to also handle mutable state by representing the heap as recursive data type. However, Birkedal and Harper's proof technique breaks down in the presence of control operators, so exceptions or first-class continuations remain a problem. Recently, Crary and Harper have built upon this work (2007).

McQueen, Plotkin, and Sethi developed a domain theoretic model for languages with polymorphism and recursive types based upon interpreting types as *ideals* in the domain (1984). Melliès and Vouillon (2005) have shown how to reformulate ideal models in a more syntactic fashion using ideas similar to Krevine's *realizablity* models (2001). This model is easily extended to provide an equivalence relation that provides a notion of parametricity, but it is unclear how complicated it might be to extend this model to language features like mutable references.

Ahmed, starting from the *step-indexed* models of Appel and McAllester, developed a proof that provides the same kind of relational reasoning provided as the parametricity theorem in the presence of iso-recursive types (2006; 2001). Step-index models represent types, τ, by pairs of indices, k, and values, v, such that for k reduction steps v approximates a value of type τ. That is, any program using v as if it had type τ can make k steps before possibly entering a stuck state. This model is appealing because it avoids much of the complicated meta-theory required by approaches like that of Pitts, Birkedal and Harper, or Melliès and Vouillon. Additionally, step-index models have been extended to handle languages with mutable state and other advanced language features (Ahmed 2004).

Recently there has been some research into proving a modified version of the parametricity theorem for languages with control operators, such as exceptions. In particular, by studying the image of a polymorphic version of Parigot's $\lambda\mu$-calculus under a continuation passing transform, Hasegawa was able to "reverse engineer" the necessary conditions for a parametricity theorem (1992; 2005). He calls the resulting property *focal parametricity*.

Finally, Sumii and Pierce have developed a coinductive bisimulation proof technique that can be used to prove contextual equivalence of programs in the presence of recursive types (2005). Unfortunately, this technique does not currently provide the same generality as is available from logical relations style or syntactic noninterference proofs: it can only show the contextual equivalence of two specific programs. Furthermore, constructing the witnessing bisimulation for the two programs can for be quite difficult, and it is unlikely that mechanically constructing such bisimulations will be possible in the near future.

§ Constructor contexts in generalized parametricity

One problem I mentioned in § 2·2 is that it is very difficult to define families of relations for use with the generalized parametricity theorem that are not parametric in the constructor context. If **Typerec** were removed from $\lambda_{\text{SEC}i}$, this problem goes away. Therefore, it is not an issue for (informally) reasoning about InforML programs, because InforML does not include a mechanism similar to **Typerec**. However, **Typerec** can be very useful, so it would be worthwhile finding a way to resolve the difficulty.

While considering this problem it struck me that it seems very similar to the problem of proving contextual equivalence directly. For contextual equivalence the problem is proving by induction over all possible program contexts that two expressions will behave the same when placed in those contexts. Usually the solution is to develop some other property for relating two expressions, and show that property is equivalent to contextual equivalence. For example, CIU-equivalence, as defined by Pitts (2005), is one such property.

The problem with defining families of relations for generalized parametricity is the need to develop a function from any possible context to a relation. Constructing such a function is isomorphic to constructing an inductive proof over contexts, so perhaps similar ideas to those used to prove contextual equivalence indirectly could be used to indirectly specify functions on constructor contexts.

Another angle on the problem with constructor contexts is that there is a mismatch between how I have extended parametricity to non-standard types and how parametricity has typically been extended to higher-kinds. Generalized parametricity quantifies over functions from labels and constructor contexts to relations. For example, I use the following notation for relations in § 2:

$$\mathsf{R}^{\ell}_{\xi} \in ((\xi\{\tau_1\}) \mathbin{@} \ell) \leftrightarrow ((\xi\{\tau_2\}) \mathbin{@} \ell),$$

The essence of R can be understood better type-theoretically as an entity with the type

$$\Pi\ell.\Pi\xi.((\xi\{\tau_1\}) \mathbin{@} \ell) \leftrightarrow ((\xi\{\tau_2\}) \mathbin{@} \ell),$$

where $\cdot \leftrightarrow \cdot$ can be understood as the "type constructor" of relations.

However, when standard parametricity is extended to higher kinds (Vytiniotis and Weirich 2007B), such as $\star \to \star$, functions from relations to relations are quantified over. For example, if ψ is a quantified type variable with kind $\star \to \star$, it would be necessary to quantify over an entity with the type

$$[\![\psi]\!] : \Pi\alpha{:}\star.\Pi\beta{:}\star.(\alpha \leftrightarrow \beta) \to (\tau_1\alpha \leftrightarrow \tau_2\beta),$$

that is, a function from an arbitrary pair of types α and β, and a relation between them, $\alpha \leftrightarrow \beta$, to the a relation $\tau_1\alpha \leftrightarrow \tau_1\beta$, for some $\tau_1 : \star \to \star$ and $\tau_2 : \star \to \star$. Furthermore, the type of this entity is completely derived from ψ's kind and the choice of τ_1 and τ_2:

$$\begin{aligned} [\![\star]\!](\tau_1,\tau_2) &\triangleq \tau_1 \leftrightarrow \tau_2 \\ [\![\kappa_1 \to \kappa_2]\!](\tau_1,\tau_2) &\triangleq \Pi\alpha{:}\kappa_1.\Pi\beta{:}\kappa_1.[\![\kappa_1]\!](\alpha,\beta) \to [\![\kappa_2]\!](\tau_1\alpha,\tau_2\beta) \end{aligned}$$

Therefore, it seems plausible that a similar approach could be used to express more interesting relationships between abstract data types in generalized parametricity. Such a solution would give R a type something like

$$\Pi\ell.\Pi\xi.[\![\xi]\!] \to ((\xi\{\tau_1\}) \mathbin{@} \ell) \leftrightarrow ((\xi\{\tau_2\}) \mathbin{@} \ell),$$

where $[\![\xi]\!]$ is some function on relations. In the case of the constructor context hole, $\bullet$, $[\![\bullet]\!]$ should most likely be the identity function on relations. Determining the definition of $[\![\cdot]\!]$ on more complex constructor contexts, and whether this is even most suitable formulation, will require further study.

I conjecture the problem with the expressive power of constructor contexts will also arise in attempts to prove parametricity like properties for other languages with expressive type systems. For example, languages with indexed and dependent type systems where type equivalence is non-parametric with respect to abstract indices or values. Therefore, I expect that having a better understanding of how to deal with constructor contexts for generalized parametricity will have much wider applicability.

§ A logical account of generative types and information-flow kinds

One aspect of InforML that is disappointing is that is conflates the information learned by using `typecase` with the information that can be learned using `toSpine`. For example, if the algebraic data type `A` has the kind `Lab -(+)-> % @ ⊤` and the type variable `α` has the kind `* @ ⊥`, the expression `typecase α of A @ l => ... end` must have an information content of `⊤`. However, `typecase` works by analyzing the structure of its scrutinee. The fact that `α` has kind `* @ ⊥` indicates that there is no information content to `α`'s structure. The reason that the entire expression must receive an information content of `⊤` is because InforML must conservatively assume that because `A @ l` has the kind `% @ ⊤` that it could learn some structural information with an information content of `⊤`. The problem is that it is a priori impossible for `typecase` to learn any structural information because the type `A` is atomic.

On the other hand, it is simply not an option to always give algebraic data types an information content of `⊥` because `toSpine` can be used to learn information about the structure of *instances* of type `A @ l`. Again, I emphasize that neither `typecase` nor `toSpine` learn anything about the structure of type `A`, because it has none.

However, when InforML was originally designed, it seemed sensible to partly tie the information learned by using `toSpine` to the kind of its input. This is, in fact, the original reason for the distinction between the kinds of algebraic data types `% @ l` and all other types `* @ l`. The idea was that the information learned from `toSpine` would be obtained from the kinds of the form `% @ l` and the information learned by `typecase` would be obtained from kinds of the form `* @ l`. Furthermore, I added a subkinding rule that stated `% @ l <: * @ ⊥`, which would account for the fact that algebraic data types have no structural content. This however, was not particularly aesthetically pleasing, and it eventually was dropped when I discovered that the subkinding rule `% @ l <: * @ ⊥` could be used to construct a covert channel – it was then replaced with the one currently used by InforML, `% @ l <: * @ l`.

Another option might have been to give algebraic kinds two labels. For example, `% @ l1 @ l2` would propagate information `l1` with `toSpine` and `l2` with `typecase`. It could then be given the subkinding rule `% @ l1 @ l2 <: * @ l2`. However, it is not clear to me that this is really the correct solution. Algebraic data types are used to combine many independent concepts in InforML, such a iso-recursive types, sum types, and generative types. Therefore, I think to better address TDP and generativity in a future revision of InforML, it will be necessary to study the problem at a more foundational level. At a logical level, all structures in a language have an introduction form and an elimination form. Information is propagated

by the use a structure's elimination form. In InforML all of this is obscured by the high level mechanisms for defining algebraic data types and pattern matching.

In a line of research mostly orthogonal to the one I present in this dissertation, I helped develop a core calculus, called $\lambda_{\mathcal{L}}$ (Vytiniotis, Washburn, and Weirich 2005), for studying type generativity and open extensibility. Unlike InforML, $\lambda_{\mathcal{L}}$ makes type generativity very explicit by providing a operation for creating "fresh" names for generative data types, and primitives terms for explicitly coercing to and from the underlying definition of a generative data type. That is much like functions or tuples, there was an explicit mechanism for introducing and eliminating generative data types. I conjecture that the information learned by `toSpine` is not associated with the kind of an algebraic data type, but is from the use of the eliminating coercion at the level of terms.

Therefore, I expect that it would be worth studying a calculus that combines $\lambda_{\mathcal{L}}$ and λ_{SECi} so that it is possible to better understand and express the distinction between `typecase` and `toSpine` in a high-level language like InforML.

§ 6·2 Conclusions

While information-flow type systems have been used in the past to provide confidentiality and integrity policies for data, I am the first to suggest lifting information-flow labeling to the kind level so that it is possible to reason about confidentiality and integrity of type meta-data (Washburn and Weirich 2005). Specifically, in this document:

- I provide a refined analysis of the problem of representation independence in the presence of TDP using the finer-grained properties of confidentiality and integrity (§ 1·2). I discussed how information-flow kind and type systems can recover the ability to reason statically about the confidentiality and integrity of ADTs as well as enforce policies on type meta-data (§ 1·3). I also explained how access control mechanisms and runtime monitoring can be applied to the problem of enforcing confidentiality and integrity policies on type meta-data, and how they compare with the use of information-flow kind and type systems.

- In order to formally verify my claims about the use of an information-flow type and kind system, I have shown how it is possible generalize the parametricity theorem so that it can be applied to languages that include runtime type analysis (§ 2). The parametricity theorem has been the primary basis for all formal reasoning about data abstraction until now (§ 2·2). I conjecture that my theorem is a straightforward generalization of the standard parametricity theorem, and show how confidentiality and integrity can be derived from my theorem as corollaries (§ 2·2). I have explained how the basis of this generalization works and have given a detailed[1] paper proof of the theorem. (§ C).

- I have described the language InforML, a realistic programming language, and explored in detail the differences between it and the language λ_{SECi} that was used to formalized generalized parametricity (§ 3·1). Additionally I have described the use of InforML's module system (§ 3·2), given a detailed

1. It is detailed for a paper proof, at least.

account of its generative algebraic data types (§ 3·3), and explained InforML's mechanisms for dynamic programming with information-flow (§ 3·4). Finally, I give a detail comparison of InforML with the language FlowCaml (§ 3·6).

- I have made a detailed exploration of how two common idioms and design patterns can be applied to realistic InforML programming: the harmless reflection idiom and the break and recover idiom. The harmless reflection idiom distinguishes between essential and essential computations and provides programmers with a guarantee that changes in the implementation of abstract data types will never affect essential computations, and that the integrity of these abstractions will never be violated (§ 4·2). The break and recover idiom trades the highly prescribed use of type-directed programming for static guarantees about how changes in representation will alter the behavior of the program, while still guaranteeing that integrity is preserved (§ 4·4).
- I have given an overview of the implementation of the InforML language, along with an review of the most significant design choices made during InforML's development (§ 5).

A

Glossary of notation

There are some semantic differences in the typefaces used in this document. Text that corresponds to code that one would directly enter into a computer is written in a sans-serif monospaced typeface. Text that corresponds to mathematical abstractions is written in a proportional typeface. For example, when discussing program text I would write `typecase` and `Bool` while discussing their mathematical abstractions I would write **typecase** and **bool**.

ℓ,ℓ	::=	(atomic) labels	Π	::=	full labels
$\mathtt{l}$	::=	label variables	π	::=	variances
κ,κ	::=	kinds	τ,τ	::=	type constructors, monotypes
α,β,ω, ...	α,β,ω, ...	::=	type variables	σ,σ	::=	types, polytypes
ρ	::=	higher-rank types	ξ	::=	constructor contexts
ν	::=	whnf type constructors	ζ	::=	whnf types
Δ	::=	type variable contexts	δ	::=	type substitutions
$\mathtt{R}$	::=	typed binary relations	η	::=	maps from type variables to relations
e,$\mathtt{e}$	::=	terms or expressions	$\mathtt{r}$	::=	record selectors
x,y,z, ...	$\mathtt{x}$,$\mathtt{y}$,$\mathtt{z}$, ...	::=	term variables	v,$\mathtt{v}$	::=	values
Γ	::=	term variable contexts	γ	::=	term substitutions
$\mathtt{A}$	::=	algebraic data types	$\mathtt{D}$	::=	data constructors
φ	::=	type patterns	$\mathtt{p}$	::=	term patterns
μ	::=	type matches	$\mathtt{u}$	::=	term matches
$\mathtt{ld}$	::=	local declarations	$\mathtt{d}$	::=	declarations
$\mathtt{M}$	::=	modules	$\mathtt{m}$	::=	module variables
$\mathtt{S}$	::=	signatures	$\mathtt{s}$	::=	signature variables
$\mathtt{sb}$	::=	signature binding			

Table A·1: Summary of meta-variables used in the document.

B

Full specification of $\lambda_{\mathbf{SEC}i}$

§ B·1 Grammar

Definition B·1·1 (Type Grammar)**.**

$$
\begin{array}{rrll}
\textit{kinds} & \kappa & ::= \star^{\ell} & \textit{types} \\
 & & \mid\ \kappa_1 \xrightarrow{\ell} \kappa_2 & \textit{operators} \\
 & & & \\
\textit{type constructors} & \tau & ::= \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2 & \lambda\textit{-calculus} \\
 & & \mid\ \mathbf{bool} & \textit{booleans} \\
 & & \mid\ \tau_1 \to \tau_2 & \textit{functions} \\
 & & \mid\ \tau_1 \times \tau_2 & \textit{products} \\
 & & \mid\ \mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} & \textit{analysis} \\
 & & & \\
\textit{whnf constructors} & \nu & ::= \xi\{\alpha\} \mid \mathbf{bool} \mid \tau_1 \xrightarrow{\ell} \tau_2 \mid \tau_1 \times^{\ell} \tau_2 \mid \lambda\alpha{:}\kappa.\tau & \\
 & & & \\
\textit{constructor contexts} & \xi & ::= \bullet \mid \mathbf{Typerec}\ \xi\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} \mid \xi\,\tau & \\
 & & & \\
\textit{types} & \sigma & ::= (\tau)\ @\ \ell & \textit{injection} \\
 & & \mid\ \sigma_1 \xrightarrow{\ell} \sigma_2 & \textit{functions} \\
 & & \mid\ \sigma_1 \times^{\ell} \sigma_2 & \textit{products} \\
 & & \mid\ \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma & \textit{polymorphism} \\
 & & & \\
\textit{whnf types} & \zeta & ::= (\mathbf{bool})\ @\ \ell \mid (\xi\{\alpha\})\ @\ \ell \mid \sigma_1 \xrightarrow{\ell} \sigma_2 \mid \sigma_1 \times^{\ell} \sigma_2 \mid \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma & \\
 & & & \\
\textit{type substitutions} & \delta & ::= \cdot \mid \delta,[\tau/\alpha] & \\
\textit{type variable contexts} & \Delta & ::= \cdot \mid \Delta,\alpha{:}\kappa &
\end{array}
$$

Definition B·1·2 (Term Grammar)**.**

$$
\begin{array}{rrll}
\textit{terms} & e & ::= & \mathbf{true} \mid \mathbf{false} & \textit{booleans} \\
& & \mid & x \mid \lambda x{:}\sigma.e \mid e_1 e_2 & \lambda\textit{-calculus} \\
& & \mid & \langle e_1, e_2 \rangle \mid \mathbf{fst}\, e \mid \mathbf{snd}\, e & \textit{tuples} \\
& & \mid & \Lambda\alpha{:}\star^{\ell}.e \mid e[\tau] & \textit{polymorphism} \\
& & \mid & \mathbf{fix}\, x{:}\sigma.e & \textit{fix-point} \\
& & \mid & \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 & \textit{conditional} \\
& & \mid & \mathbf{typecase}[\gamma.\sigma]\, \tau\, e_{\mathbf{bool}}\, e_{\rightarrow}\, e_{\times} & \textit{analysis}
\end{array}
$$

$$
\textit{values} \quad v ::= \mathbf{true} \mid \mathbf{false} \mid \lambda x{:}\sigma.e \mid \langle v_1, v_2 \rangle \mid \Lambda\alpha{:}\star^{\ell}.e
$$

$$
\begin{array}{rl}
\textit{term substitutions} & \gamma ::= \cdot \mid \gamma, [e/x] \\
\textit{term variable contexts} & \Gamma ::= \cdot \mid \Gamma, x{:}\sigma
\end{array}
$$

§ B·2 Kind and type label operators

$$
\begin{array}{lll}
\text{Kind information} & \mathcal{L}(\star^{\ell}) \triangleq \ell & \mathcal{L}(\kappa_1 \xrightarrow{\ell} \kappa_2) \triangleq \ell \\
\text{Kind join} & \star^{\ell_1} \sqcup \ell_2 \triangleq \star^{(\ell_1 \sqcup \ell_2)} & (\kappa_1 \xrightarrow{\ell_1} \kappa_2) \sqcup \ell_2 \triangleq \kappa_1 \xrightarrow{\ell_1 \sqcup \ell_2} \kappa_2 \\
\text{Type information} & \mathcal{L}((\tau) @ \ell) \triangleq \ell & \mathcal{L}(\sigma_1 \xrightarrow{\ell} \sigma_2) \triangleq \ell \\
 & \mathcal{L}(\sigma_1 \times^{\ell} \sigma_2) \triangleq \ell & \mathcal{L}(\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma) \triangleq \ell_1 \\
\text{Type join} & (\tau) @ \ell_1 \sqcup \ell_2 \triangleq (\tau) @ (\ell_1 \sqcup \ell_2) & (\sigma_1 \xrightarrow{\ell_1} \sigma_2) \sqcup \ell_2 \triangleq \sigma_1 \xrightarrow{\ell_1 \sqcup \ell_2} \sigma_2 \\
 & (\sigma_1 \times^{\ell_1} \sigma_2) \sqcup \ell_2 \triangleq \sigma_1 \times^{(\ell_1 \sqcup \ell_2)} \sigma_2 & (\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma) \sqcup \ell_3 \triangleq \forall^{(\ell_1 \sqcup \ell_3)}\alpha{:}\star^{\ell_2}.\sigma
\end{array}
$$

§ B·3 Static semantics

Definition B·3·1 (Sub-kinding)**.**

$$
\frac{}{\kappa \leq \kappa}\ \textsc{sbk:refl} \qquad \frac{\kappa_1 \leq \kappa_2 \qquad \kappa_2 \leq \kappa_3}{\kappa_1 \leq \kappa_3}\ \textsc{sbk:trans} \qquad \frac{\ell_1 \sqsubseteq \ell_2}{\star^{\ell_1} \leq \star^{\ell_2}}\ \textsc{sbk:type}
$$

$$
\frac{\kappa_3 \leq \kappa_1 \qquad \kappa_2 \leq \kappa_4 \qquad \ell_1 \sqsubseteq \ell_2}{\kappa_1 \xrightarrow{\ell_1} \kappa_2 \leq \kappa_3 \xrightarrow{\ell_2} \kappa_4}\ \textsc{sbk:arr}
$$

Definition B·3·2 (Constructor well-formedness)**.**

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsc{wfc:var} \qquad \frac{}{\Delta \vdash \mathbf{bool} : \star^{\bot}}\ \textsc{wfc:bool} \qquad \frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 : \star^{\ell_2}}{\Delta \vdash \tau_1 \to \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{wfc:arr}$$

$$\frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 : \star^{\ell_2}}{\Delta \vdash \tau_1 \times \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{wfc:prod} \qquad \frac{\Delta, \alpha{:}\kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda \alpha{:}\kappa_1.\tau : \kappa_1 \xrightarrow{\bot} \kappa_2}\ \textsc{wfc:abs}$$

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 : \kappa_2 \sqcup \ell}\ \textsc{wfc:app}$$

$$\frac{\begin{array}{l}\Delta \vdash \tau : \star^{\ell} \qquad \Delta \vdash \tau_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\mathbf{bool}} : \kappa \qquad \Delta \vdash \tau_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa\end{array} \quad \text{where } \ell' = \mathcal{L}(\kappa) \text{ and } \ell \sqsubseteq \ell'}{\Delta \vdash \mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa}\ \textsc{wfc:trec}$$

$$\frac{\Delta \vdash \tau : \kappa_1 \qquad \kappa_1 \leq \kappa_2}{\Delta \vdash \tau : \kappa_2}\ \textsc{wfc:sub}$$

Definition B·3·3 (Constructor equivalence)**.**

$$\frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \tau = \tau : \kappa}\ \textsc{eqc:refl} \qquad \frac{\Delta \vdash \tau_1 = \tau_2 : \kappa \qquad \Delta \vdash \tau_2 = \tau_3 : \kappa}{\Delta \vdash \tau_1 = \tau_3 : \kappa}\ \textsc{eqc:trans} \qquad \frac{\Delta \vdash \tau_2 = \tau_1 : \kappa}{\Delta \vdash \tau_1 = \tau_2 : \kappa}\ \textsc{eqc:sym}$$

$$\frac{\Delta \vdash \tau_3 = \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 = \tau_4 : \star^{\ell_2}}{\Delta \vdash \tau_1 \to \tau_2 = \tau_3 \to \tau_4 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{eqc:arr} \qquad \frac{\Delta \vdash \tau_3 = \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 = \tau_4 : \star^{\ell_2}}{\Delta \vdash \tau_1 \times \tau_2 = \tau_3 \times \tau_4 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{eqc:prod}$$

$$\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau_1 = \tau_2 : \kappa_2}{\Delta \vdash \lambda \alpha{:}\kappa_1.\tau_1 = \lambda \alpha{:}\kappa_1.\tau_2 : \kappa_1 \xrightarrow{\bot} \kappa_2}\ \textsc{eqc:abs-con} \qquad \frac{\Delta \vdash (\lambda \alpha{:}\kappa_1.\tau_1)\tau_2 : \kappa_2}{\Delta \vdash (\lambda \alpha{:}\kappa_1.\tau_1)\tau_2 = \tau_1[\tau_2/\alpha] : \kappa_2}\ \textsc{eqc:abs-beta}$$

$$\frac{\Delta \vdash \tau_1 = \tau_3 : \kappa_1 \xrightarrow{\ell} \kappa_2 \qquad \Delta \vdash \tau_2 = \tau_3 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 = \tau_3 \tau_4 : \kappa_2 \sqcup \ell}\ \textsc{eqc:app}$$

$$\frac{\begin{array}{l}\Delta \vdash \tau_1 = \tau_2 : \star^{\ell} \qquad \Delta \vdash \tau_{\mathbf{bool}} = \tau'_{\mathbf{bool}} : \kappa \qquad \Delta \vdash \tau_{\to} = \tau'_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\times} = \tau'_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \qquad \ell \sqsubseteq \ell' \qquad \text{where } \ell' = \mathcal{L}(\kappa)\end{array}}{\begin{array}{l}\Delta \vdash \mathbf{Typerec}\ \tau_1 \quad = \mathbf{Typerec}\ \tau_2 \quad : \kappa \\ \qquad \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} \qquad \tau'_{\mathbf{bool}}\ \tau'_{\to}\ \tau'_{\times}\end{array}}\ \textsc{eqc:trec-con}$$

$$\frac{\Delta \vdash \mathbf{Typerec\ bool}\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa}{\Delta \vdash \mathbf{Typerec\ bool}\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} = \tau_{\mathbf{bool}} : \kappa}\ \textsc{eqc:trec-bool}$$

$$\frac{\Delta \vdash \mathbf{Typerec}\ (\tau_1 \to \tau_2)\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa}{\Delta \vdash \mathbf{Typerec}\ (\tau_1 \to \tau_2)\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} = \tau_{\to}\tau_1\tau_2\ (\mathbf{Typerec}\ \tau_1\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times})\ (\mathbf{Typerec}\ \tau_2\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times}) : \kappa}\ \textsc{eqc:trec-arr}$$

$$\frac{\Delta \vdash \mathbf{Typerec}\ (\tau_1 \times \tau_2)\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa}{\Delta \vdash \mathbf{Typerec}\ (\tau_1 \times \tau_2)\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} = \tau_{\times}\tau_1\tau_2\ (\mathbf{Typerec}\ \tau_1\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times})\ (\mathbf{Typerec}\ \tau_2\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times}) : \kappa}\ \textsc{eqc:trec-prod}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 : \kappa_1 \qquad \kappa_1 \le \kappa_2}{\Delta \vdash \tau_1 = \tau_2 : \kappa_2}\ \textsc{eqc:sub}$$

Definition B·3·4 (Type variable context restriction)**.** *We will write $\Delta^\star$ for those type variable contexts Δ where $\forall \alpha{:}\kappa \in \Delta$, $\kappa = \star^\ell$ for some ℓ.*

Definition B·3·5 (Subtyping)**.**

$$\frac{\Delta^\star \vdash \sigma}{\Delta^\star \vdash \sigma \le \sigma}\ \textsc{sbt:refl} \qquad \frac{\Delta^\star \vdash \sigma_1 \le \sigma_2 \qquad \Delta^\star \vdash \sigma_2 \le \sigma_3}{\Delta^\star \vdash \sigma_1 \le \sigma_3}\ \textsc{sbt:trans}$$

$$\frac{\Delta^\star \vdash \tau_1 = \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1)\ @\ \ell_2 \le (\tau_2)\ @\ \ell_2}\ \textsc{sbt:con} \qquad \frac{\Delta^\star \vdash \tau_1 \to \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1 \to \tau_2)\ @\ \ell_2 \le (\tau_1)\ @\ \ell_2 \xrightarrow{\ell_2} (\tau_2)\ @\ \ell_2}\ \textsc{sbt:con-arr1}$$

$$\frac{\Delta^\star \vdash \tau_1 \to \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1)\ @\ \ell_2 \xrightarrow{\ell_2} (\tau_2)\ @\ \ell_2 \le (\tau_1 \to \tau_2)\ @\ \ell_2}\ \textsc{sbt:con-arr2}$$

$$\frac{\Delta^\star \vdash \tau_1 \times \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1 \times \tau_2)\ @\ \ell_2 \le (\tau_1)\ @\ \ell_2 \times^{\ell_2} (\tau_2)\ @\ \ell_2}\ \textsc{sbt:con-prod1}$$

$$\frac{\Delta^\star \vdash \tau_1 \times \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1)\ @\ \ell_2 \times^{\ell_2} (\tau_2)\ @\ \ell_2 \le (\tau_1 \times \tau_2)\ @\ \ell_2}\ \textsc{sbt:con-prod2}$$

$$\frac{\Delta^\star \vdash \sigma_3 \le \sigma_1 \qquad \Delta^\star \vdash \sigma_2 \le \sigma_4 \qquad \ell_1 \sqsubseteq \ell_2}{\Delta^\star \vdash \sigma_1 \xrightarrow{\ell_1} \sigma_2 \le \sigma_3 \xrightarrow{\ell_2} \sigma_4}\ \textsc{sbt:arr}$$

$$\frac{\Delta^\star \vdash \sigma_1 \le \sigma_3 \qquad \Delta^\star \vdash \sigma_2 \le \sigma_4 \qquad \ell_1 \sqsubseteq \ell_2}{\Delta^\star \vdash \sigma_1 \times^{\ell_1} \sigma_2 \le \sigma_3 \times^{\ell_2} \sigma_4}\ \textsc{sbt:prod}$$

$$\frac{\Delta^\star,\alpha{:}\star^{\ell_4} \vdash \sigma_1 \le \sigma_2 \qquad \ell_4 \sqsubseteq \ell_2 \qquad \ell_1 \sqsubseteq \ell_3}{\Delta^\star \vdash \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma_1 \le \forall^{\ell_3}\alpha{:}\star^{\ell_4}.\sigma_2}\ \textsc{sbt:all}$$

Definition B·3·6 (Type well-formedness)**.**

$$\frac{\Delta^\star \vdash \tau : \star^{\ell_1}}{\Delta^\star \vdash (\tau)\ @\ \ell_2}\ \textsc{wftp:con} \qquad \frac{\Delta^\star \vdash \sigma_1 \qquad \Delta^\star \vdash \sigma_2}{\Delta^\star \vdash \sigma_1 \xrightarrow{\ell} \sigma_2}\ \textsc{wftp:arr} \qquad \frac{\Delta^\star \vdash \sigma_1 \qquad \Delta^\star \vdash \sigma_2}{\Delta^\star \vdash \sigma_1 \times^{\ell} \sigma_2}\ \textsc{wftp:prod}$$

$$\frac{\Delta^\star,\alpha{:}\star^{\ell_1} \vdash \sigma}{\Delta^\star \vdash \forall^{\ell_2}\alpha{:}\star^{\ell_1}.\sigma}\ \textsc{wftp:all}$$

Definition B·3·7 (Type equivalence)**.** *We define* $\Delta^\star \vdash \sigma_1 = \sigma_2$ *to mean that* $\Delta^\star \vdash \sigma_1 \le \sigma_2$ *and* $\Delta^\star \vdash \sigma_2 \le \sigma_1$*.*

Definition B·3·8 (Term variable context well-formedness)**.**

$$\frac{}{\Delta^\star \vdash \cdot}\ \textsc{wftc:empty} \qquad \frac{\Delta^\star \vdash \Gamma \qquad \Delta^\star \vdash \sigma}{\Delta^\star \vdash \Gamma, x{:}\sigma}\ \textsc{wftc:cons}$$

Definition B·3·9 (Term well-formedness)**.**

$$\frac{\Delta^\star \vdash \Gamma}{\Delta^\star;\Gamma \vdash \mathbf{true} : (\mathbf{bool})\ @\ \bot}\ \textsc{wft:true} \qquad \frac{\Delta^\star \vdash \Gamma}{\Delta^\star;\Gamma \vdash \mathbf{false} : (\mathbf{bool})\ @\ \bot}\ \textsc{wft:false}$$

$$\frac{\Delta^\star \vdash \Gamma \qquad x : \sigma \in \Gamma}{\Delta^\star;\Gamma \vdash x : \sigma}\ \textsc{wft:var} \qquad \frac{\Delta^\star;\Gamma,x{:}\sigma_1 \vdash e : \sigma_2 \qquad \Delta^\star \vdash \sigma_1}{\Delta^\star;\Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \xrightarrow{\bot} \sigma_2}\ \textsc{wft:abs}$$

$$\frac{\Delta^\star;\Gamma \vdash e_1 : \sigma_1 \xrightarrow{\ell} \sigma_2 \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma_1}{\Delta^\star;\Gamma \vdash e_1 e_2 : \sigma_2 \sqcup \ell}\ \textsc{wft:app} \qquad \frac{\Delta^\star,\alpha{:}\star^{\ell};\Gamma \vdash e : \sigma}{\Delta^\star;\Gamma \vdash \Lambda\alpha{:}\star^{\ell}.e : \forall^{\bot}\alpha{:}\star^{\ell}.\sigma}\ \textsc{wft:tabs}$$

$$\frac{\Delta^\star;\Gamma \vdash e : \forall^{\ell}\alpha{:}\star^{\ell'}.\sigma \qquad \Delta^\star \vdash \tau : \star^{\ell'}}{\Delta^\star;\Gamma \vdash e[\tau] : \sigma[\tau/\alpha] \sqcup \ell}\ \textsc{wft:tapp} \qquad \frac{\Delta^\star;\Gamma \vdash e_1 : \sigma_1 \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma_2}{\Delta^\star;\Gamma \vdash \langle e_1,e_2\rangle : \sigma_1 \times^{\bot} \sigma_2}\ \textsc{wft:pair}$$

$$\frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2}{\Delta^\star;\Gamma \vdash \mathbf{fst}\, e : \sigma_1 \sqcup \ell}\ \textsc{wft:fst} \qquad \frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2}{\Delta^\star;\Gamma \vdash \mathbf{snd}\, e : \sigma_2 \sqcup \ell}\ \textsc{wft:snd} \qquad \frac{\Delta^\star;\Gamma,x{:}\sigma \vdash e : \sigma \qquad \Delta^\star \vdash \sigma}{\Delta^\star;\Gamma \vdash \mathbf{fix}\ x{:}\sigma.e : \sigma}\ \textsc{wft:fix}$$

$$\frac{\Delta^\star;\Gamma \vdash e_1 : (\mathbf{bool})\ @\ \ell \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma \qquad \Delta^\star;\Gamma \vdash e_3 : \sigma}{\Delta^\star;\Gamma \vdash \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 : \sigma \sqcup \ell}\ \textsc{wft:if}$$

$$\frac{\begin{array}{cc}\Delta^\star \vdash \tau : \star^\ell & \Delta^\star;\Gamma \vdash e_{\mathbf{bool}} : \sigma[\mathbf{bool}/\gamma] \\ \Delta^\star,\gamma{:}\star^\ell \vdash \sigma & \Delta^\star;\Gamma \vdash e_\to : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \to \beta/\gamma] \qquad \text{where } \ell' = \mathcal{L}(\sigma[\tau/\gamma]) \\ \ell \sqsubseteq \ell' & \Delta^\star;\Gamma \vdash e_\times : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \times \beta/\gamma]\end{array}}{\Delta^\star;\Gamma \vdash \mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathbf{bool}}\ e_\to\ e_\times : \sigma[\tau/\gamma]}\ \textsc{wft:tcase}$$

$$\frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \qquad \Delta^\star \vdash \sigma_1 \leq \sigma_2}{\Delta^\star;\Gamma \vdash e : \sigma_2}\ \textsc{wft:sub}$$

§ B·4 Dynamic semantics

Definition B·4·1 (Constructor reduction)**.**

$$\frac{\tau_1 \rightsquigarrow \tau_1'}{\tau_1\tau_2 \rightsquigarrow \tau_1'\tau_2}\ \textsc{whr:app-con} \qquad \frac{}{(\lambda\alpha{:}\kappa.\tau_1)\tau_2 \rightsquigarrow \tau_1[\tau_2/\alpha]}\ \textsc{whr:app}$$

$$\frac{\tau \rightsquigarrow \tau'}{\mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times \rightsquigarrow \mathbf{Typerec}\ \tau'\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times}\ \textsc{whr:trec-con}$$

$$\frac{}{\mathbf{Typerec}\ (\mathbf{bool})\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times \rightsquigarrow \tau_{\mathbf{bool}}}\ \textsc{whr:trec-bool}$$

$$\frac{}{\begin{array}{l}\mathbf{Typerec}\ (\tau_1 \to \tau_2)\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times \rightsquigarrow \tau_\to\ \tau_1\ \tau_2\ (\mathbf{Typerec}\ \tau_1\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times) \\ \qquad\qquad (\mathbf{Typerec}\ \tau_2\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times)\end{array}}\ \textsc{whr:trec-arr}$$

$$\frac{}{\begin{array}{l}\mathbf{Typerec}\ (\tau_1 \times \tau_2)\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times \rightsquigarrow \tau_\times\ \tau_1\ \tau_2\ (\mathbf{Typerec}\ \tau_1\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times) \\ \qquad\qquad (\mathbf{Typerec}\ \tau_2\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times)\end{array}}\ \textsc{whr:trec-prod}$$

Definition B·4·2 (Term computation rules)**.**

$$\frac{}{(\lambda x{:}\sigma.e)v \rightsquigarrow e[v/x]}\ \textsc{ev:app} \qquad \frac{}{(\Lambda\alpha{:}\kappa.e)[\tau] \rightsquigarrow e[\tau/\alpha]}\ \textsc{ev:tapp} \qquad \frac{}{\mathbf{fst}\,\langle v_1,v_2\rangle \rightsquigarrow v_1}\ \textsc{ev:fst}$$

$$\frac{}{\mathbf{snd}\,\langle v_1,v_2\rangle \rightsquigarrow v_2}\ \textsc{ev:snd} \qquad \frac{}{\mathbf{fix}\ x{:}\sigma.e \rightsquigarrow e[\mathbf{fix}\ x{:}\sigma.e/x]}\ \textsc{ev:fix} \qquad \frac{}{\mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow e_1}\ \textsc{ev:if1}$$

$$\frac{}{\mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow e_2}\ \textsc{ev:if2} \qquad \frac{\tau \rightsquigarrow^* \mathbf{bool}}{\mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathbf{bool}}\ e_\to\ e_\times \rightsquigarrow e_{\mathbf{bool}}}\ \textsc{ev:tcase-bool}$$

$$\frac{\tau \rightsquigarrow^* \tau_1 \to \tau_2}{\textbf{typecase}\,[\gamma.\sigma]\,\tau\,e_{\textbf{bool}}\,e_{\to}\,e_{\times} \rightsquigarrow e_{\to}[\tau_1][\tau_2]}\ \textsc{ev:tcase-arr}$$

$$\frac{\tau \rightsquigarrow^* \tau_1 \times \tau_2}{\textbf{typecase}\,[\gamma.\sigma]\,\tau\,e_{\textbf{bool}}\,e_{\to}\,e_{\times} \rightsquigarrow e_{\times}[\tau_1][\tau_2]}\ \textsc{ev:tcase-prod}$$

Definition B·4·3 (Term congruence rules)**.**

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 e_2 \rightsquigarrow e_1' e_2}\ \textsc{ev:app1} \qquad \frac{e_2 \rightsquigarrow e_2'}{v_1 e_2 \rightsquigarrow v_1 e_2'}\ \textsc{ev:app2} \qquad \frac{e_1 \rightsquigarrow e_1'}{\langle e_1, e_2\rangle \rightsquigarrow \langle e_1', e_2\rangle}\ \textsc{ev:pair1}$$

$$\frac{e_2 \rightsquigarrow e_2'}{\langle v_1, e_2\rangle \rightsquigarrow \langle v_1, e_2'\rangle}\ \textsc{ev:pair2} \qquad \frac{e \rightsquigarrow e'}{\textbf{fst}\,e \rightsquigarrow \textbf{fst}\,e'}\ \textsc{ev:fst-con} \qquad \frac{e \rightsquigarrow e'}{\textbf{snd}\,e \rightsquigarrow \textbf{snd}\,e'}\ \textsc{ev:snd-con}$$

$$\frac{e_1 \rightsquigarrow e_1'}{\textbf{if}\,e_1\,\textbf{then}\,e_2\,\textbf{else}\,e_3 \rightsquigarrow \textbf{if}\,e_1'\,\textbf{then}\,e_2\,\textbf{else}\,e_3}\ \textsc{ev:if-con} \qquad \frac{e \rightsquigarrow e'}{e[\tau] \rightsquigarrow e'[\tau]}\ \textsc{ev:tapp-con}$$

Definition B·4·4 (Nontermination)**.** *If* $\cdot;\cdot \vdash e : \sigma$ *and there does not exist a derivation* $e \rightsquigarrow^* v$ *then* $e \uparrow$*.*

C

Generalized parametricity for $\lambda_{\mathbf{SEC}i}$

§ C·1 Soundness

Lemma C·1·1 (Inversion on sub-kinding)**.**

1. *If* $\star^{\ell} \leq \kappa$ *then* $\kappa = \star^{\ell'}$ *where* $\ell \sqsubseteq \ell'$*.*
2. *If* $\kappa_1 \xrightarrow{\ell} \kappa_2 \leq \kappa$ *then* $\kappa = \kappa_3 \xrightarrow{\ell'} \kappa_4$ *where* $\kappa_3 \leq \kappa_1$ *and* $\kappa_2 \leq \kappa_4$ *and* $\ell \sqsubseteq \ell'$*.*

Proof. Straightforward induction over the structure of the sub-kinding derivation. □

Lemma C·1·2 (Inversion for constructor well-formedness)**.**

1. *If* $\Delta \vdash \tau_1 \to \tau_2 : \star^{\ell}$ *then* $\Delta \vdash \tau_1 : \star^{\ell_1}$ *and* $\Delta \vdash \tau_2 : \star^{\ell_2}$ *and* $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$*.*
2. *If* $\Delta \vdash \tau_1 \times \tau_2 : \star^{\ell}$ *then* $\Delta \vdash \tau_1 : \star^{\ell_1}$ *and* $\Delta \vdash \tau_2 : \star^{\ell_2}$ *and* $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$*.*
3. *If* $\Delta \vdash \tau_1\tau_2 : \kappa$ *then* $\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2$ *and* $\Delta \vdash \tau_2 : \kappa_1$ *and* $\kappa_2 \sqcup \ell \leq \kappa$*.*
4. *If* $\Delta \vdash \lambda\alpha{:}\kappa.\tau : \kappa_1 \xrightarrow{\ell} \kappa_2$ *then* $\Delta,\alpha{:}\kappa \vdash \tau : \kappa_3$ *and* $\kappa_1 \leq \kappa$ *and* $\kappa_3 \leq \kappa_2$*.*
5. *If* $\Delta \vdash \mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa$ *then* $\Delta \vdash \tau : \star^{\ell}$ *and* $\Delta \vdash \tau_{\mathbf{bool}} : \kappa'$ *and* $\Delta \vdash \tau_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa' \xrightarrow{\ell'} \kappa' \xrightarrow{\ell'} \kappa'$ *and* $\Delta \vdash \tau_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa' \xrightarrow{\ell'} \kappa' \xrightarrow{\ell'} \kappa'$ *where* $\ell' = \mathcal{L}(\kappa')$ *and* $\kappa' \leq \kappa$*.*

Proof. By induction over the structure of the well-formedness derivation, making use of Lemma C·1·1. □

Lemma C·1·3 (Weak-head reduction equivalence)**.**

1. *If* $\Delta \vdash \tau : \kappa$ *and* $\tau \rightsquigarrow \tau'$ *then* $\Delta \vdash \tau = \tau' : \kappa$*.*

2. *If* $\Delta \vdash \tau : \kappa$ *and* $\tau \rightsquigarrow^* \tau'$ *then* $\Delta \vdash \tau = \tau' : \kappa$.

3. *If* $\Delta^\star \vdash \sigma$ *and* $\sigma \rightsquigarrow \sigma'$ *then* $\Delta^\star \vdash \sigma = \sigma'$.

4. *If* $\Delta^\star \vdash \sigma$ *and* $\sigma \rightsquigarrow^* \sigma'$ *then* $\Delta^\star \vdash \sigma = \sigma'$.

Proof. Part 1 follows from straightforward induction over the structure of $\tau \rightsquigarrow \tau'$ and use of Lemma C·1·2. Part 2 follows from Part 1 and induction on the number of reduction steps. Part 3 follows from straightforward induction over the structure of $\sigma \rightsquigarrow \sigma'$ using Part 1. Finally, Part 4 follows from Part 3 and induction on the number of reduction steps. □

Lemma C·1·4 (Inversion for type well-formedness)**.**

If $\Delta^\star \vdash (\tau) @ \ell$ *then* $\Delta^\star \vdash \tau : \star^{\ell'}$.

Proof. Proof by induction over the structure of $\Delta^\star \vdash (\tau) @ \ell$. □

Lemma C·1·5 (Inversion for subtyping)**.**

1. *If* $\Delta^\star \vdash \sigma_1 \xrightarrow{\ell_1} \sigma_2 \leq \sigma$ *then* $\Delta^\star \vdash \sigma \leq \sigma_3 \xrightarrow{\ell_2} \sigma_4$ *and* $\Delta^\star \vdash \sigma_3 \leq \sigma_1$ *and* $\Delta^\star \vdash \sigma_2 \leq \sigma_4$ *and* $\ell_1 \sqsubseteq \ell_2$.

2. *If* $\Delta^\star \vdash \sigma_1 \times^{\ell_1} \sigma_2 \leq \sigma$ *then* $\Delta^\star \vdash \sigma \leq \sigma_3 \times^{\ell_2} \sigma_4$ *and* $\Delta^\star \vdash \sigma_1 \leq \sigma_3$ *and* $\Delta^\star \vdash \sigma_2 \leq \sigma_4$ *and* $\ell_1 \sqsubseteq \ell_2$.

3. *If* $\Delta^\star \vdash \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma_1 \leq \sigma_2$ *then* $\Delta^\star \vdash \sigma_2 \leq \forall^{\ell_3}\alpha{:}\star^{\ell_4}.\sigma_3$ *and* $\Delta^\star,\alpha{:}\star^{\ell_4} \vdash \sigma_3 \leq \sigma_1$ *and* $\ell_1 \sqsubseteq \ell_3$ *and* $\ell_4 \sqsubseteq \ell_2$.

Proof. By straightforward induction over the structure of the subtyping derivation. □

Lemma C·1·6 (Inversion for typing)**.**

1. *If* $\Delta^\star;\Gamma \vdash \lambda x{:}\sigma_1.e : \sigma$ *then* $\Delta^\star \vdash \sigma \leq \sigma_2 \xrightarrow{\ell} \sigma_3$ *and* $\Delta^\star;\Gamma,x{:}\sigma_1 \vdash e : \sigma_4$ *where* $\Delta^\star \vdash \sigma_2 \leq \sigma_1$ *and* $\Delta^\star \vdash \sigma_4 \leq \sigma_3$.

2. *If* $\Delta^\star;\Gamma \vdash \Lambda\alpha{:}\star^{\ell}.e : \sigma$ *then* $\Delta^\star \vdash \sigma \leq \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma_1$ *and* $\Delta^\star,\alpha{:}\star^{\ell};\Gamma \vdash e : \sigma_2$ *where* $\Delta^\star,\alpha{:}\star^{\ell_2} \vdash \sigma_2 \leq \sigma_1$ *and* $\ell_2 \sqsubseteq \ell$.

3. *If* $\Delta^\star;\Gamma \vdash \mathbf{fix}\ x{:}\sigma_1.e : \sigma_2$ *then* $\Delta^\star;\Gamma,x{:}\sigma_1 \vdash e : \sigma_1$ *where* $\Delta^\star \vdash \sigma_1 \leq \sigma_2$.

4. *If* $\Delta^\star;\Gamma \vdash \langle e_1,e_2\rangle : \sigma$ *then* $\Delta^\star \vdash \sigma \leq \sigma_1 \times^{\ell} \sigma_2$ *and* $\Delta^\star;\Gamma \vdash e_1 : \sigma_3$ *and* $\Delta^\star;\Gamma \vdash e_2 : \sigma_4$ *where* $\Delta^\star \vdash \sigma_3 \leq \sigma_1$ *and* $\Delta^\star \vdash \sigma_4 \leq \sigma_2$.

5. *If* $\Delta^\star;\Gamma \vdash \mathbf{fst}\, e : \sigma$ *then* $\Delta^\star;\Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2$ *where* $\Delta^\star \vdash \sigma_1 \sqcup \ell \leq \sigma$.

6. *If* $\Delta^\star;\Gamma \vdash \mathbf{snd}\, e : \sigma$ *then* $\Delta^\star;\Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2$ *where* $\Delta^\star \vdash \sigma_2 \sqcup \ell \leq \sigma$.

7. *If* $\Delta^\star;\Gamma \vdash e_1e_2 : \sigma_1$ *then* $\Delta^\star;\Gamma \vdash e_1 : \sigma_2 \xrightarrow{\ell} \sigma_3$ *and* $\Delta^\star;\Gamma \vdash e_2 : \sigma_2$ *and* $\Delta^\star \vdash \sigma_3 \sqcup \ell \leq \sigma_1$.

8. *If* $\Delta^\star;\Gamma \vdash e[\tau] : \sigma$ *then* $\Delta^\star;\Gamma \vdash e : \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma'$ *and* $\Delta^\star \vdash \tau : \star^{\ell_2}$ *and* $\Delta^\star \vdash \sigma'[\tau/\alpha] \sqcup \ell_1 \leq \sigma$.

9. *If* $\Delta^\star;\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \sigma$ *then* $\Delta^\star;\Gamma \vdash e_1 : (\mathbf{bool})\ @\ \ell$ *and* $\Delta^\star;\Gamma \vdash e_2 : \sigma'$ *and* $\Delta^\star;\Gamma \vdash e_3 : \sigma'$ *where* $\Delta^\star \vdash \sigma' \sqcup \ell \leq \sigma$.

10. *If* $\Delta^\star;\Gamma \vdash \mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathbf{bool}}\ e_{\rightarrow}\ e_{\times} : \sigma'$ *then*
$\Delta^\star \vdash \tau : \star^\ell$ *and*
$\Delta^\star,\gamma{:}\star^\ell \vdash \sigma$ *and*
$\Delta^\star;\Gamma \vdash e_{\mathbf{bool}} : \sigma[\mathbf{bool}/\gamma]$ *and*
$\Delta^\star;\Gamma \vdash e_{\rightarrow} : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \rightarrow \beta/\gamma]$ *and*
$\Delta^\star;\Gamma \vdash e_{\times} : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \times \beta/\gamma]$ *where*
$\ell' = \mathcal{L}(\sigma[\tau/\gamma])$ *and*
$\ell \sqsubseteq \ell'$ *and*
$\Delta^\star \vdash \sigma[\tau/\gamma] \leq \sigma'$.

Proof. By straightforward induction on the structure of the typing derivation with uses of Lemma C·1·5. □

Lemma C·1·7 (Substitution for constructors)**.** *If* $\Delta,\alpha{:}\kappa_1 \vdash \tau_1 : \kappa_2$ *and* $\Delta \vdash \tau_2 : \kappa_1$ *then* $\Delta \vdash \tau_1[\tau_2/\alpha] : \kappa_2$.

Proof. By straightforward induction over the structure of $\Delta,\alpha{:}\kappa_1 \vdash \tau_1 : \kappa_2$. □

Lemma C·1·8 (Substitution for equivalence)**.** *If* $\Delta,\alpha{:}\kappa_1 \vdash \tau_1 = \tau_2 : \kappa_2$ *and* $\Delta \vdash \tau : \kappa_1$ *then* $\Delta \vdash \tau_1[\tau/\alpha] = \tau_2[\tau/\alpha] : \kappa_2$.

Proof. By straightforward induction over the structure of $\Delta,\alpha{:}\kappa_1 \vdash \tau_1 = \tau_2 : \kappa_2$, making use of Lemma C·1·7. □

Lemma C·1·9 (Substitution for types)**.**

1. *If* $\Delta^\star,\alpha{:}\star^\ell \vdash \sigma_1 \leq \sigma_2$ *and* $\Delta^\star \vdash \tau : \star^\ell$ *then* $\Delta^\star \vdash \sigma_1[\tau/\alpha] \leq \sigma_2[\tau/\alpha]$.
2. *If* $\Delta^\star,\alpha{:}\star^\ell \vdash \sigma$ *and* $\Delta^\star \vdash \tau : \star^\ell$ *then* $\Delta^\star \vdash \sigma[\tau/\alpha]$.

Proof. By mutual induction over the structure of $\Delta,\alpha{:}\star^\ell \vdash \sigma_1 \leq \sigma_2$ and $\Delta,\alpha{:}\star^\ell \vdash \sigma$, using Lemmas C·1·7 and C·1·8. □

Lemma C·1·10 (Substitution commutes with equivalence)**.**

1. *If* $\Delta \vdash \tau_1 = \tau_2 : \kappa_1$ *and* $\Delta,\alpha{:}\kappa_1 \vdash \tau : \kappa_2$ *then* $\Delta \vdash \tau[\tau_1/\alpha] = \tau[\tau_2/\alpha] : \kappa_2$.
2. *If* $\Delta \vdash \tau_1 = \tau_2 : \star^\ell$ *and* $\Delta,\alpha{:}\star^\ell \vdash \sigma$ *then* $\Delta \vdash \sigma[\tau_1/\alpha] \leq \sigma[\tau_2/\alpha]$ *and* $\Delta \vdash \sigma[\tau_2/\alpha] \leq \sigma[\tau_1/\alpha]$.

Proof. Part 1 follows from induction over the structure of $\Delta,\alpha{:}\kappa_1 \vdash \tau : \kappa_2$. Part 2 follows from induction over the structure of $\Delta,\alpha{:}\star^\ell \vdash \sigma$ making use of Part 1. □

Lemma C·1·11 (Substitution for terms)**.**

1. *If* $\Delta^\star,\alpha{:}\star^\ell;\Gamma \vdash e : \sigma$ *and* $\Delta^\star \vdash \tau : \star^\ell$ *then* $\Delta^\star;\Gamma \vdash e[\tau/\alpha] : \sigma[\tau/\alpha]$.
2. *If* $\Delta^\star;\Gamma,x{:}\sigma_1 \vdash e : \sigma_2$ *and* $\Delta^\star;\Gamma \vdash e' : \sigma_1$ *then* $\Delta^\star;\Gamma \vdash e[e'/x] : \sigma_2$.

Proof. By straightforward induction over the typing derivations, using Lemmas C·1·7 and C·1·9. □

Lemma C·1·12 (Subject reduction)**.**

1. *If* $\Delta \vdash \tau : \kappa$ *and* $\tau \rightsquigarrow \tau'$ *then* $\Delta \vdash \tau' : \kappa$.
2. *If* $\Delta \vdash \tau : \kappa$ *and* $\tau \rightsquigarrow^* \tau'$ *then* $\Delta \vdash \tau' : \kappa$.
3. *If* $\Delta^\star \vdash \sigma$ *and* $\sigma \rightsquigarrow \sigma'$ *then* $\Delta^\star \vdash \sigma'$.
4. *If* $\Delta^\star \vdash \sigma$ *and* $\sigma \rightsquigarrow^* \sigma'$ *then* $\Delta^\star \vdash \sigma'$.
5. *If* $\Delta^\star; \Gamma \vdash e : \sigma$ *and* $e \rightsquigarrow e'$ *then* $\Delta^\star; \Gamma \vdash e' : \sigma$.

Proof. Part 1 follows by induction over the structure of $\tau \rightsquigarrow \tau'$ making use of Lemmas C·1·2 and C·1·7. Part 2 is a direct corollary of Part 1. Part 3 follows by induction over the structure of $\sigma \rightsquigarrow \sigma'$ making use of Lemma C·1·4 and Part 1. Part 4 is a direct corollary of Part 3. Part 5 follows by induction over the structure of $e \rightsquigarrow e'$ making use of Lemmas C·1·6, C·1·2, C·1·3, and C·1·10. □

Lemma C·1·13 (Weak head reduction terminates)**.**

1. *If* $\cdot \vdash \tau : \kappa$ *then* $\tau \rightsquigarrow^* \nu$.
2. *If* $\Delta^\star \vdash \sigma$ *then* $\sigma \rightsquigarrow^* \zeta$.

Proof. Follows from a standard logical relations proof that we omit here. See Morrisett's thesis (Morrisett 1995). □

Lemma C·1·14 (Canonical forms for constructors)**.** *If* $\cdot \vdash \nu : \kappa$

1. $\kappa = \star^\ell$ *then* $\nu = \mathbf{bool}$ *or* $\nu = \tau_1 \to \tau_2$ *or* $\nu = \tau_1 \times \tau_2$.
2. $\kappa = \kappa_1 \xrightarrow{\ell} \kappa_2$ *then* $\nu = \lambda\alpha{:}\kappa_3.\tau$ *where* $\kappa_1 \leq \kappa_3$.

Proof. By straightforward induction over the structure of $\Delta \vdash \nu : \kappa$. □

Lemma C·1·15 (Canonical forms for terms)**.** *If* $\cdot; \cdot \vdash \nu : \sigma$

1. $\sigma = \mathbf{bool}$ *then* $\nu = \mathbf{true}$ *or* $\nu = \mathbf{false}$.
2. $\sigma = \sigma_1 \xrightarrow{\ell'} \sigma_2$ *then* $\nu = \lambda x{:}\sigma_3.e$ *where* $\Delta^\star \vdash \sigma_1 \leq \sigma_3$.
3. $\sigma = \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma'$ *then* $\nu = \Lambda\alpha{:}\star^{\ell_3}.e$ *where* $\ell_1 \sqsubseteq \ell_3$.
4. $\sigma = \sigma_1 \times^\ell \sigma_2$ *then* $\nu = \langle \nu_1, \nu_2 \rangle$.

Proof. By straightforward induction over the structure of $\cdot; \cdot \vdash \nu : \sigma$. □

Lemma C·1·16 (Progress)**.** *If* $\cdot; \cdot \vdash e : \sigma$ *then* e *is a value or there exists a derivation* $e \rightsquigarrow e'$.

Proof. By straightforward induction over the structure of $\cdot;\cdot \vdash e : \sigma$, using Lemmas C·1·15, C·1·13, and C·1·14. □

Theorem C·1·17 (Type safety)**.** *If* $\cdot;\cdot \vdash e : \sigma$ *then there exists a derivation that* $e \rightsquigarrow^* v$ *or* $e \uparrow$.

Proof. Proof by contradiction using Lemmas C·1·12 and C·1·16. □

§ C·2 Finite unwindings

Definition C·2·1 (Extension for finite unwindings)**.**

$$\begin{array}{rcll} \textit{terms}\ e & ::= & \ldots & \\ & \mid & \mathbf{fix}_n\ x{:}\sigma.e & \textit{finite fix-point} \end{array}$$

Definition C·2·2 (Term well-formedness)**.**

$$\frac{\Delta^\star;\Gamma,x{:}\sigma \vdash e : \sigma \qquad \Delta^\star \vdash \sigma}{\Delta^\star;\Gamma \vdash \mathbf{fix}_n\ x{:}\sigma.e : \sigma}\ \textsc{wft:fixn}$$

Definition C·2·3 (Computation rules)**.**

$$\frac{}{\mathbf{fix}_0\ x{:}\sigma.e \rightsquigarrow \mathbf{fix}_0\ x{:}\sigma.e}\ \textsc{ev:fix0} \qquad \frac{}{\mathbf{fix}_{n+1}\ x{:}\sigma.e \rightsquigarrow e[\mathbf{fix}_n\ x{:}\sigma.e/x]}\ \textsc{ev:fixn}$$

Definition C·2·4 (Annotation erasure)**.** *Given a term* e *another term* e'*,* e' *is an "erasure" of* e *the inductive relation* $e \preceq e'$ *holds.*

$$\frac{}{\mathbf{true} \preceq \mathbf{true}}\ \textsc{ler:true} \qquad \frac{}{\mathbf{false} \preceq \mathbf{false}}\ \textsc{ler:false} \qquad \frac{}{x \preceq x}\ \textsc{ler:var} \qquad \frac{e_1 \preceq e_2}{\lambda x{:}\sigma.e_1 \preceq \lambda x{:}\sigma.e_2}\ \textsc{ler:abs}$$

$$\frac{e_1 \preceq e_3 \qquad e_2 \preceq e_4}{e_1 e_2 \preceq e_3 e_4}\ \textsc{ler:app} \qquad \frac{e_1 \preceq e_3 \qquad e_2 \preceq e_4}{\langle e_1,e_2\rangle \preceq \langle e_3,e_4\rangle}\ \textsc{ler:pair} \qquad \frac{e_1 \preceq e_2}{\mathbf{fst}\, e_1 \preceq \mathbf{fst}\, e_2}\ \textsc{ler:fst}$$

$$\frac{e_1 \preceq e_2}{\mathbf{snd}\, e_1 \preceq \mathbf{snd}\, e_2}\ \textsc{ler:snd} \qquad \frac{e_1 \preceq e_2}{\Lambda\alpha{:}\star^\ell.e_1 \preceq \Lambda\alpha{:}\star^\ell.e_2}\ \textsc{ler:tabs} \qquad \frac{e_1 \preceq e_2}{e_1[\tau] \preceq e_2[\tau]}\ \textsc{ler:tapp}$$

$$\frac{e_1 \preceq e_2}{\mathbf{fix}\ x{:}\sigma.e_1 \preceq \mathbf{fix}\ x{:}\sigma.e_2}\ \textsc{ler:fix} \qquad \frac{e_1 \preceq e_2}{\mathbf{fix}_n\ x{:}\sigma.e_1 \preceq \mathbf{fix}_n\ x{:}\sigma.e_2}\ \textsc{ler:fixn1}$$

$$\frac{e_1 \preceq e_2}{\mathbf{fix}_n\ x{:}\sigma.e_1 \preceq \mathbf{fix}\ x{:}\sigma.e_2}\ \textsc{ler:fixn2} \qquad \frac{e_1 \preceq e_4 \qquad e_2 \preceq e_5 \qquad e_3 \preceq e_6}{\mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 \preceq \mathbf{if}\, e_4\, \mathbf{then}\, e_5\, \mathbf{else}\, e_6}\ \textsc{ler:if}$$

$$\frac{e_{\mathbf{bool}} \preceq e'_{\mathbf{bool}} \qquad e_\to \preceq e'_\to \qquad e_\times \preceq e'_\times}{\mathbf{typecase}[\gamma.\sigma]\,\tau\, e_{\mathbf{bool}}\, e_\to\, e_\times \preceq \mathbf{typecase}[\gamma.\sigma]\,\tau\, e'_{\mathbf{bool}}\, e'_\to\, e'_\times}\ \textsc{ler:tcase}$$

Lemma C·2·5 ($\mathbf{fix}_o$ always diverges)**.** $\mathbf{fix}_o\ x{:}\sigma.e \uparrow$.

Proof. Proof by contradiction, assuming there exists a derivation $\mathbf{fix}_o\ x{:}\sigma.e \rightsquigarrow^* v$. □

Lemma C·2·6 (Unwinding type equivalences)**.**

$$\Delta^\star; \Gamma \vdash \mathbf{fix}\ x{:}\sigma.e : \sigma \quad \textit{iff} \quad \Delta^\star; \Gamma \vdash \mathbf{fix}_n\ x{:}\sigma.e : \sigma$$

Proof. Trivial inversion upon the typing derivation in both directions. □

Lemma C·2·7 (Unwinding evaluation equivalence)**.**

$\mathbf{fix}\ x{:}\sigma.e' \rightsquigarrow^* v$ *iff exists* n *such that for all* $m, m \geq n$ *and* $\mathbf{fix}_m\ x{:}\sigma.e' \rightsquigarrow^* v'$ *where* $v' \preceq v$

Proof. Both directions follow by straightforward induction over number or reduction steps. □

§ C·3 Noninterference

Definition C·3·1 (Relations between values)**.** *We define* $\sigma_1 \leftrightarrow \sigma_2$ *to be the set of all binary relations between values of type* σ_1 *and values of type* σ_2*.*

Definition C·3·2 (Parameterized relation)**.** *A parameterized relation* R *is a function that when given a label* ℓ *and a type context* ρ *yields a binary relation between values of two types. For conciseness, we use the notation* R^ℓ_ρ *for the application of a label and a type context to a parameterized relation.*

We will sometimes abuse notation and write

$$R^\ell_\rho \in \delta_1((\rho\{\tau_1\}) \mathbin{@} \ell) \leftrightarrow \delta_2((\rho\{\tau_2\}) \mathbin{@} \ell).$$

This can be roughly understood with dependent types as

$$R : \Pi\ell.\Pi\rho.\delta_1((\rho\{\tau_1\}) \mathbin{@} \ell) \leftrightarrow \delta_2((\rho\{\tau_2\}) \mathbin{@} \ell).$$

Definition C·3·3 (Parameterized relation consistency)**.** *We say that a parameterized relation* $R^\ell_\rho \in \sigma_1 \leftrightarrow \sigma_2$ *is consistent if*

1. $v_1 R^{\ell_1}_\rho v_2$ *and* $\ell_1 \sqsubseteq \ell_2$ *then* $v_1 R^{\ell_2}_\rho v_2$ *(moving up in the lattice does not change relatedness)*
2. $v_1 \preceq v_2$ *and* $v_3 \preceq v_4$ *and* $v_1 R^{\ell_1}_\rho v_3$ *then* $v_2 R^{\ell_1}_\rho v_4$ *(relation does not treat finite approximations differently)*

Definition C·3·4 (Security logical relation for constructors)**.**

$$\frac{\ell_1 \not\sqsubseteq \ell_o}{v_1 \sim_{\ell_o} v_2 : \star^{\ell_1}}\ \textsc{tslr:type-opaq} \qquad \frac{\ell_1 \sqsubseteq \ell_o}{\mathbf{bool} \sim_{\ell_o} \mathbf{bool} : \star^{\ell_1}}\ \textsc{tslr:type-bool}$$

$$\frac{\ell_1 \sqcup \ell_2 \sqsubseteq \ell_3 \qquad \ell_3 \sqsubseteq \ell_o \qquad \tau_1 \approx_{\ell_o} \tau_3 : \star^{\ell_1} \qquad \tau_2 \approx_{\ell_o} \tau_4 : \star^{\ell_2}}{\tau_1 \to \tau_2 \sim_{\ell_o} \tau_3 \to \tau_4 : \star^{\ell_3}}\ \textsc{tslr:type-arr}$$

$$\frac{\ell_1 \sqcup \ell_2 \sqsubseteq \ell_3 \qquad \ell_3 \sqsubseteq \ell_o \qquad \tau_1 \approx_{\ell_o} \tau_3 : \star^{\ell_1} \qquad \tau_2 \approx_{\ell_o} \tau_4 : \star^{\ell_2}}{\tau_1 \times \tau_2 \sim_{\ell_o} \tau_3 \times \tau_4 : \star^{\ell_3}}\ \textsc{tslr:type-prod}$$

$$\frac{\forall(\tau_1 \approx_{\ell_o} \tau_2 : \kappa_1).\nu_1\tau_1 \approx_{\ell_o} \nu_2\tau_2 : \kappa_2 \sqcup \ell_1}{\nu_1 \sim_{\ell_o} \nu_2 : \kappa_1 \xrightarrow{\ell_1} \kappa_2}\ \textsc{tslr:arr}$$

$$\frac{\tau_1 \rightsquigarrow^* \nu_1 \qquad \tau_2 \rightsquigarrow^* \nu_2 \qquad \nu_1 \sim_{\ell_o} \nu_2 : \kappa}{\tau_1 \approx_{\ell_o} \tau_2 : \kappa}\ \textsc{tsclr:base}$$

We implicitly require for $\nu_1 \sim_\ell \nu_2 : \kappa$ *and* $\tau_1 \approx_\ell \tau_2 : \kappa$ *that* $\cdot \vdash \nu_1,\nu_2 : \kappa$ *and* $\cdot \vdash \tau_1,\tau_2 : \kappa$ *respectively.*

Definition c·3·5 (Type reduction)**.**

$$\frac{\tau \rightsquigarrow \tau'}{(\tau)\ @\ \ell \rightsquigarrow (\tau')\ @\ \ell}\ \textsc{whr:inj-tc} \qquad \frac{}{(\tau_1 \to \tau_2)\ @\ \ell \rightsquigarrow (\tau_1)\ @\ \ell \xrightarrow{\ell} (\tau_2)\ @\ \ell}\ \textsc{whr:inj-arr}$$

$$\frac{}{(\tau_1 \times \tau_2)\ @\ \ell \rightsquigarrow (\tau_1)\ @\ \ell \times^{\ell} (\tau_2)\ @\ \ell}\ \textsc{whr:inj-prod}$$

Definition c·3·6 (Security logical relation for terms)**.**

$$\frac{\alpha \mapsto \mathrm{R} \in \eta \qquad (\ell_1 \sqsubseteq \ell_o) \Longrightarrow (\nu_1 \mathrm{R}_\xi^{\ell_1} \nu_2)}{\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : (\xi\{\alpha\})\ @\ \ell_1}\ \textsc{slr:con} \qquad \frac{(\ell_1 \sqsubseteq \ell_o) \Longrightarrow (\nu_1 = \nu_2)}{\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : (\mathbf{bool})\ @\ \ell_1}\ \textsc{slr:bool}$$

$$\frac{\forall(\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma_1).\eta \vdash \nu_1 e_1 \approx_{\ell_o} \nu_2 e_2 : \sigma_2 \sqcup \ell_1}{\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \sigma_1 \xrightarrow{\ell_1} \sigma_2}\ \textsc{slr:arr}$$

$$\frac{\eta \vdash \mathbf{fst}\ \nu_1 \approx_{\ell_o} \mathbf{fst}\ \nu_2 : \sigma_1 \sqcup \ell_1 \qquad \eta \vdash \mathbf{snd}\ \nu_1 \approx_{\ell_o} \mathbf{snd}\ \nu_2 : \sigma_2 \sqcup \ell_1}{\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \sigma_1 \times^{\ell_1} \sigma_2}\ \textsc{slr:prod}$$

$$\frac{\begin{array}{c}\forall(\tau_1 \approx_{\ell_o} \tau_2 : \star^{\ell_2}).\forall(\mathrm{R}_\xi^{\ell_2} \in \delta_1((\xi\{\tau_1\})\ @\ \ell_2) \leftrightarrow \delta_2((\xi\{\tau_2\})\ @\ \ell_2)).\\ \eta,\alpha \mapsto \mathrm{R} \vdash \nu_1[\tau_1] \approx_{\ell_o} \nu_2[\tau_2] : \sigma \sqcup \ell_1 \qquad \mathrm{R}\ \text{consistent}\end{array}}{\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma}\ \textsc{slr:all}$$

$$\frac{e_1 \rightsquigarrow^* \nu_1 \qquad e_2 \rightsquigarrow^* \nu_2 \qquad \sigma \rightsquigarrow^* \zeta \qquad \eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta}{\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma}\ \textsc{sclr:term} \qquad \frac{e_1 \uparrow}{\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma}\ \textsc{sclr:divr1}$$

$$\frac{e_2 \uparrow}{\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma}\ \textsc{sclr:divr2}$$

We implicitly require for $\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta$ *and* $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma$ *that* $\cdot;\cdot \vdash \nu_1 : \delta_1(\zeta)$*,* $\cdot;\cdot \vdash \nu_2 : \delta_2(\zeta)$ *and* $\cdot;\cdot \vdash e_1 : \delta_1(\sigma)$*,* $\cdot;\cdot \vdash e_2 : \delta_2(\sigma)$ *respectively where* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *and* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$*.*

Definition C·3·7 (Related constructor substitutions)**.**

$$\frac{\forall \alpha{:}\kappa \in \Delta.(\delta_1(\alpha) \approx_{\ell_o} \delta_2(\alpha) : \kappa)}{\delta_1 \approx_{\ell_o} \delta_2 : \Delta} \text{ TSSLR:BASE}$$

Definition C·3·8 (Relation mapping regularity)**.** *If* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *then*

$$\frac{\forall \alpha{:}\star^{\ell_1} \in \Delta^\star.(\eta(\alpha)^{\ell_1}_{\xi} \in \delta_1((\xi\{\alpha\}) \mathbin{@} \ell_1) \leftrightarrow \delta_2((\xi\{\alpha\}) \mathbin{@} \ell_1)) \qquad \eta(\alpha) \text{ consistent}}{\delta_1,\delta_2 \vdash \eta : \Delta^\star} \text{ RELM:REG}$$

Definition C·3·9 (Related term substitutions)**.** *If* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *and* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *then*

$$\frac{\forall x{:}\sigma \in \Gamma.(\eta \vdash \gamma_1(x) \approx_{\ell_o} \gamma_2(x) : \sigma)}{\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma} \text{ SSLR:BASE}$$

Lemma C·3·10 (Logical relations are closed under reduction)**.**

1. $\tau_1 \approx_{\ell_o} \tau_2 : \kappa$ *iff* $\tau_1 \rightsquigarrow^* \tau_1'$ *and* $\tau_2 \rightsquigarrow^* \tau_2'$ *and* $\tau_1' \approx_{\ell_o} \tau_2' : \kappa$.
2. $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma$ *iff* $e_1 \rightsquigarrow^* e_1'$ *and* $e_2 \rightsquigarrow^* e_2'$ *and* $\sigma \rightsquigarrow^* \sigma'$ *and* $\eta \vdash e_1' \approx_{\ell_o} e_2' : \sigma'$.

Proof. Follows from straightforward inversion upon the logical relations and from the properties of reduction. □

Lemma C·3·11 (Inversion for subtyping on normal types)**.**

1. *If* $\Delta^\star \vdash (\rho\{\alpha\}) \mathbin{@} \ell_1 \leq \zeta$ *then* $\zeta = (\rho\{\alpha\}) \mathbin{@} \ell_2$ *where* $\ell_1 \sqsubseteq \ell_2$.
2. *If* $\Delta^\star \vdash (\mathbf{bool}) \mathbin{@} \ell_1 \leq \zeta$ *then* $\zeta = (\mathbf{bool}) \mathbin{@} \ell_2$ *where* $\ell_1 \sqsubseteq \ell_2$.

Proof. By straightforward induction over the structure of the subtyping derivations. □

Lemma C·3·12 (Logical relations are closed under erasure)**.**

1. *If* $v_1' \preceq v_1$ *and* $v_2' \preceq v_2$ *and* $\eta \vdash v_1' \sim_{\ell_o} v_2' : \zeta$ *then* $\eta \vdash v_1 \sim_{\ell_o} v_2 : \zeta$.
2. *If* $e_1' \preceq e_1$ *and* $e_2' \preceq e_2$ *and* $\eta \vdash e_1' \approx_{\ell_o} e_2' : \sigma$ *then* $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma$

Proof. The proof of Parts 1 and 2 follows by straightforward mutual induction over the structure of $\eta \vdash v_1' \sim_{\ell_o} v_2' : \zeta$ and $\eta \vdash e_1' \approx_{\ell_o} e_2' : \sigma$. □

Lemma C·3·13 (Logical relations are closed under subsumption)**.**

1. *If* $\kappa_1 \leq \kappa_2$ *and*

- $\nu_1 \sim_{\ell_o} \nu_2 : \kappa_1$ *then* $\tau_1 \sim_{\ell_o} \tau_2 : \kappa_2$.
- $\tau_1 \approx_{\ell_o} \tau_2 : \kappa_1$ *then* $\tau_1 \approx_{\ell_o} \tau_2 : \kappa_2$

2. *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and*
   - $\Delta^\star \vdash \zeta_1 \leq \zeta_2$ *and* $\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_1$ *then* $\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_2$.
   - $\Delta^\star \vdash \sigma_1 \leq \sigma_2$ *and* $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma_1$ *then* $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma_2$

Proof. Part 1 follows from straightforward mutual induction over κ_1. Part 2 follows from straightforward mutual induction over σ_1 and ζ_1, with uses of Part 1, Definition C·3·3, and Lemmas C·1·5 and C·3·11. □

Corollary C·3·14 (Value relation is consistent)**.** *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and* $\Delta^\star \vdash \tau : \star^\top$*, then the relation*

$$\mathrm{R}^{\rho}_{\ell} = \{(\nu_1,\nu_2) \;\middle|\; \eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : (\rho\{\tau\}) \mathbin{@} \ell\}$$

is consistent.

Proof. A direct consequence of Definition C·3·3, Lemma C·3·13 Part 2, and C·3·12 Part 2. □

Lemma C·3·15 (Obliviousness)**.**

1. *If* $\cdot \vdash \tau_1,\tau_2 : \kappa$ *and* $\mathcal{L}(\kappa) \not\sqsubseteq \ell_o$ *then* $\tau_1 \approx_{\ell_o} \tau_2 : \kappa$.
2. *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *and* $\mathcal{L}(\zeta) \not\sqsubseteq \ell_o$ *and*
   - $\Delta^\star;\cdot \vdash \nu_1,\nu_2 : \zeta$ *then* $\eta \vdash \delta_1(\nu_1) \sim_{\ell_o} \delta_2(\nu_2) : \zeta$.
   - $\Delta^\star;\cdot \vdash e_1,e_2 : \sigma$ *then* $\eta \vdash \delta_1(e_1) \approx_{\ell_o} \delta_2(e_2) : \sigma$.

Proof. Part 1 follows from the use of Lemma C·1·13 and straightforward induction upon κ. Part 2 follows from Theorem C·1·17 and induction upon ζ. □

Lemma C·3·16 (Constructor substitution for term relations)**.** *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and* $\mathrm{R}^{\ell}_{\rho} = \{(\nu_1,\nu_2) \mid \eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_2\}$ *and* $\delta_i(\alpha) = \delta_i(\tau)$ *then*

1. $\eta,\alpha \mapsto \mathrm{R} \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_1$ *and* $(\rho\{\tau\}) \mathbin{@} \ell \rightsquigarrow^* \zeta_2$ *iff* $\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_3$ *where* $\zeta[\tau/\alpha] \rightsquigarrow^* \zeta_3$.
2. $\eta,\alpha \mapsto \mathrm{R} \vdash e_1 \approx_{\ell_o} e_2 : \sigma$ *and* $(\rho\{\tau\}) \mathbin{@} \ell \rightsquigarrow^* \zeta$ *iff* $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma[\tau/\alpha]$.

Proof. Follows from mutual induction over the logical relations, making use of Lemma C·3·15 Part 2 and Corollary C·3·14. □

Lemma C·3·17 (Constructor relation closed under **Typerec**)**.** *If* $\tau \approx_{\ell_o} \tau' : \star^\ell$ *and*

- $\tau_{\mathsf{bool}} \approx_{\ell_o} \tau'_{\mathsf{bool}} : \kappa$ *and*
- $\tau_{\to} \approx_{\ell_o} \tau'_{\to} : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$ *and*

- $\tau_\times \approx_{\ell_o} \tau'_\times : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$.

where $\ell' = \mathcal{L}(\kappa)$ *then* $\mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times \approx_{\ell_o} \mathbf{Typerec}\ \tau'\ \tau'_{\mathbf{bool}}\ \tau'_\to\ \tau'_\times : \kappa$.

Proof. Straightforward induction over the structure of $\tau \approx_{\ell_1} \tau' : \star^\ell$ making use of Lemma C·3·15 Part 1. □

Lemma C·3·18 (Fixpoint continuity)**.** *If for all* n, $\eta \vdash \mathbf{fix}_n\ x{:}\sigma_1.e_1 \approx_{\ell_o} \mathbf{fix}_n\ x{:}\sigma_2.e_2 : \sigma$ *then* $\eta \vdash \mathbf{fix}\ x{:}\sigma_1.e_1 \approx_{\ell_o} \mathbf{fix}\ x{:}\sigma_2.e_2 : \sigma$ *where* $\delta_i(\sigma) = \sigma_i$.

Proof. By substitution we know that $\cdot;\cdot \vdash \mathbf{fix}\ x{:}\sigma_i.e_i : \sigma_i$. Using Theorem C·1·17 we know that either $\mathbf{fix}\ x{:}\sigma_i.e_i \rightsquigarrow^* v_i$ or $\mathbf{fix}\ x{:}\sigma_i.e_i \uparrow$.

Case If both $\mathbf{fix}\ x{:}\sigma_1.e_1 \rightsquigarrow^* v_1$ and $\mathbf{fix}\ x{:}\sigma_2.e_2 \rightsquigarrow^* v_2$

- From Lemma C·2·7 we know that there is some m such that $\mathbf{fix}_m\ x{:}\sigma_1.e_1 \rightsquigarrow^* v'_i$ where $v'_i \preceq v_i$.
- Instantiating for all n, $\eta \vdash \mathbf{fix}_n\ x{:}\sigma_1.e_1 \approx_{\ell_o} \mathbf{fix}_n\ x{:}\sigma_2.e_2 : \sigma$ with m we have that $\eta \vdash \mathbf{fix}_m\ x{:}\sigma_1.e_1 \approx_{\ell_o} \mathbf{fix}_m\ x{:}\sigma_2.e_2 : \sigma$.
- By inversion upon $\eta \vdash \mathbf{fix}_m\ x{:}\sigma_1.e_1 \approx_{\ell_o} \mathbf{fix}_m\ x{:}\sigma_2.e_2 : \sigma$. we know that either $\mathbf{fix}_m\ x{:}\sigma_i.e_i \rightsquigarrow^* v'_i$ or $\mathbf{fix}_m\ x{:}\sigma_i.e_i \uparrow v'_i$. However, we already have that $\mathbf{fix}_m\ x{:}\sigma_i.e_i \rightsquigarrow^* v'_i$. Therefore, we also know by inversion that $\eta \vdash v'_1 \sim_{\ell_o} v'_2 : \zeta$ for $\sigma \rightsquigarrow^* \zeta$.
- By Lemma C·3·12 Part 1 on $v'_i \preceq v_i$ and $\eta \vdash v'_1 \sim_{\ell_o} v'_2 : \zeta$ we have that $\eta \vdash v_1 \sim_{\ell_o} v_2 : \zeta$.
- Given that $\mathbf{fix}\ x{:}\sigma_i.e_i \rightsquigarrow^* v_i$ and $\sigma \rightsquigarrow^* \zeta$ by SCLR:TERM we can conclude that $\eta \vdash \mathbf{fix}\ x{:}\sigma_1.e_1 \approx_{\ell_o} \mathbf{fix}\ x{:}\sigma_2.e_2 : \sigma$.

Case If $\mathbf{fix}\ x{:}\sigma_1.e_1 \uparrow$

- Follows directly from SCLR:DIVR1

Case If $\mathbf{fix}\ x{:}\sigma_2.e_2 \uparrow$

- Follows directly from SCLR:DIVR2

□

Theorem C·3·19 (Substitution)**.**

1. *If* $\Delta \vdash \tau : \kappa$ *and* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ *then* $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \kappa$.
2. *If* $\Delta^\star;\Gamma \vdash e : \sigma$ *and* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *and* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and* $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ *then* $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma$.

Proof. Part 1 follows by induction over the structure of $\Delta \vdash \tau : \kappa$.

Case

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \text{WFC:VAR}$$

- Immediate by inversion upon $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$.

Case

$$\frac{}{\Delta \vdash \textbf{bool} : \star^{\bot}}\ \textsc{wfc:bool}$$

- By the definition of substitution $\delta_i(\textbf{bool}) = \textbf{bool}$, and $\textbf{bool} \rightsquigarrow^* \textbf{bool}$ by TRC:REFL, therefore $\delta_i(\textbf{bool}) \rightsquigarrow^* \delta_i(\textbf{bool})$.
- $\bot \sqsubseteq \ell_o$ for any ℓ_o so it follows trivially from TSLR:TYPE-BOOL that $\textbf{bool} \sim_{\ell_o} \textbf{bool} : \star^{\bot}$.
- By TSCLR:BASE on $\textbf{bool} \sim_{\ell_o} \textbf{bool} : \star^{\bot}$ and $\delta_i(\textbf{bool}) \rightsquigarrow^* \delta_i(\textbf{bool})$ we can conclude that $\textbf{bool} \approx_{\ell_o} \textbf{bool} : \star^{\bot}$.

Case

$$\frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 : \star^{\ell_2}}{\Delta \vdash \tau_1 \to \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{wfc:arr}$$

- By the definition of substitution $\delta_i(\tau_1 \to \tau_2) = \delta_i(\tau_1) \to \delta_i(\tau_2)$ and $\delta_i(\tau_1) \to \delta_i(\tau_2) \rightsquigarrow^* \delta_i(\tau_1) \to \delta_i(\tau_2)$, by TRC:REFL, therefore $\delta_i(\tau_1 \to \tau_2) \rightsquigarrow^* \delta_i(\tau_1 \to \tau_2)$.
- Lattice joins and order are decidable, so either $\ell_1 \sqcup \ell_2 \sqsubseteq \ell_o$ or $\ell_1 \sqcup \ell_2 \not\sqsubseteq \ell_o$.

  Sub-Case $\ell_1 \sqcup \ell_2 \sqsubseteq \ell_o$.

  - Appeal to the induction hypothesis on $\Delta \vdash \tau_1 : \star^{\ell_1}$ and $\Delta \vdash \tau_2 : \star^{\ell_2}$ with $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ yielding $\delta_1(\tau_1) \approx_{\ell_o} \delta_2(\tau_1) : \star^{\ell_1}$ and $\delta_1(\tau_2) \approx_{\ell_o} \delta_2(\tau_2) : \star^{\ell_2}$.
  - Using TSLR:TYPE-ARR on these along with $\ell_1 \sqcup \ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$ (by reflexivity) and $\ell_1 \sqcup \ell_2 \sqsubseteq \ell_o$ yields
    $$\delta_1(\tau_1) \to \delta_1(\tau_2) \sim_{\ell_o} \delta_2(\tau_1) \to \delta_2(\tau_2) : \star^{\ell_1 \sqcup \ell_2}$$

  Sub-Case $\ell_1 \sqcup \ell_2 \not\sqsubseteq \ell_o$

  - It follows trivially from TSLR:TYPE-OPAQ that
    $$\delta_1(\tau_1) \to \delta_1(\tau_2) \sim_{\ell_o} \delta_2(\tau_1) \to \delta_2(\tau_2) : \star^{\ell_1 \sqcup \ell_2}$$

- Using TSCLR:BASE on $\delta_i(\tau_1) \to \delta_i(\tau_2) \rightsquigarrow^* \delta_i(\tau_1) \to \delta_i(\tau_2)$ and
  $$\delta_1(\tau_1) \to \delta_1(\tau_2) \sim_{\ell_o} \delta_2(\tau_1) \to \delta_2(\tau_2) : \star^{\ell_1 \sqcup \ell_2}$$
  gives us
  $$\delta_1(\tau_1) \to \delta_1(\tau_2) \approx_{\ell_o} \delta_2(\tau_1) \to \delta_2(\tau_2) : \star^{\ell_1 \sqcup \ell_2}$$
  which by the equality described above, is the same as
  $$\delta_1(\tau_1 \to \tau_2) \approx_{\ell_o} \delta_2(\tau_1 \to \tau_2) : \star^{\ell_1 \sqcup \ell_2}$$

Case The case for WFC:PROD is symmetric to the case for WFC:ARR.

Case

$$\frac{\Delta,\alpha{:}\kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau : \kappa_1 \xrightarrow{\perp} \kappa_2}\ \textsc{wfc:abs}$$

- By the definition of substitution $\delta_i(\lambda\alpha{:}\kappa_1.\tau) = \lambda\alpha{:}\kappa_1.\delta_i(\tau)$ and by TRC:REFL we know $\lambda\alpha{:}\kappa_1.\delta_i(\tau) \rightsquigarrow^* \lambda\alpha{:}\kappa_1.\delta_i(\tau)$, therefore $\delta_i(\lambda\alpha{:}\kappa_1.\tau) \rightsquigarrow^* \delta_i(\lambda\alpha{:}\kappa_1.\tau)$.
- Assume $\tau_1 \approx_{\ell_o} \tau_2 : \kappa_1$. Therefore, $\delta_1,[\tau_1/\alpha] \approx_{\ell_o} \delta_2,[\tau_2/\alpha] : \Delta,\alpha{:}\kappa_1$ by Definition C·3·7 and inversion upon $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$.
- Appealing to the induction hypothesis on $\Delta,\alpha{:}\kappa_1 \vdash \tau : \kappa_2$ with $\delta_1,[\tau_1/\alpha] \approx_{\ell_o} \delta_2,[\tau_2/\alpha] : \Delta,\alpha{:}\kappa_1$ we have that

  $$(\delta_1,[\tau_1/\alpha])(\tau) \approx_{\ell_o} (\delta_2,[\tau_2/\alpha])(\tau) : \kappa_2$$

- By Lemma C·3·10 we know that this is the same as

  $$(\lambda\alpha{:}\kappa_1.\delta_1(\tau))\tau_1 \approx_{\ell_o} (\lambda\alpha{:}\kappa_1.\delta_2(\tau))\tau_2 : \kappa_2$$

  Furthermore by Lemma C·3·13 Part 1 on $\kappa_2 \sqsubseteq \kappa_2 \sqcup \perp$ and

  $$(\lambda\alpha{:}\kappa_1.\delta_1(\tau))\tau_1 \approx_{\ell_o} (\lambda\alpha{:}\kappa_1.\delta_2(\tau))\tau_2 : \kappa_2$$

  we know that

  $$(\lambda\alpha{:}\kappa_1.\delta_1(\tau))\tau_1 \approx_{\ell_o} (\lambda\alpha{:}\kappa_1.\delta_2(\tau))\tau_2 : \kappa_2 \sqcup \perp$$

- Consequently, discharging our assumption we have that

  $$\lambda\alpha{:}\kappa_1.\delta_1(\tau) \sim_{\ell_o} \lambda\alpha{:}\kappa_1.\delta_2(\tau) : \kappa_1 \xrightarrow{\perp} \kappa_2$$

  Use of TSCLR:BASE on this and $\lambda\alpha{:}\kappa_1.\delta_i(\tau) \rightsquigarrow^* \lambda\alpha{:}\kappa_1.\delta_i(\tau)$ yields

  $$\lambda\alpha{:}\kappa_1.\delta_1(\tau) \approx_{\ell_o} \lambda\alpha{:}\kappa_1.\delta_2(\tau) : \kappa_1 \xrightarrow{\perp} \kappa_2$$

  By the above identity, this is the same as

  $$\delta_1(\lambda\alpha{:}\kappa_1.\tau) \approx_{\ell_o} \delta_2(\lambda\alpha{:}\kappa_1.\tau) : \kappa_1 \xrightarrow{\perp} \kappa_2$$

Case

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1\tau_2 : \kappa_2 \sqcup \ell}\ \textsc{wfc:app}$$

- Appealing to the induction hypothesis on $\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2$ and $\Delta \vdash \tau_2 : \kappa_1$ with $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ gives us $\delta_1(\tau_1) \approx_{\ell_o} \delta_2(\tau_1) : \kappa_1 \xrightarrow{\ell} \kappa_2$ and $\delta_1(\tau_2) \approx_{\ell_o} \delta_2(\tau_2) : \kappa_1$.
- By inversion upon $\delta_1(\tau_1) \approx_{\ell_o} \delta_2(\tau_1) : \kappa_1 \xrightarrow{\ell} \kappa_2$ we have that $\delta_i(\tau_1) \rightsquigarrow^* \nu_i$ and $\nu_1 \sim_{\ell_o} \nu_2 : \kappa_1 \xrightarrow{\ell} \kappa_2$. By further inversion upon $\nu_1 \sim_{\ell_o} \nu_2 : \kappa_1 \xrightarrow{\ell} \kappa_2$ we know that

  $$\forall(\tau_1' \approx_{\ell_o} \tau_2' : \kappa_1).\nu_1\tau_1' \approx_{\ell_o} \nu_2\tau_2' : \kappa_2 \sqcup \ell$$

- Instantiating this with $\delta_1(\tau_2) \approx_{\ell_o} \delta_2(\tau_2) : \kappa_1$ gives us

  $$\nu_1(\delta_1(\tau_2)) \approx_{\ell_o} \nu_2(\delta_2(\tau_2)) : \kappa_2 \sqcup \ell$$

  By inversion on this we get that $\nu_i(\delta_i(\tau_2)) \rightsquigarrow^* \nu_i'$ and $\nu_1' \sim_{\ell_o} \nu_2' : \kappa_2 \sqcup \ell_2$.
- Given $\delta_i(\tau_1) \rightsquigarrow^* \nu_i$ and $\nu_i(\delta_i(\tau_2)) \rightsquigarrow^* \nu_i'$ we know that $\delta_i(\tau_1)\delta_i(\tau_2) \rightsquigarrow^* \nu_i'$. As $\delta_i(\tau_1)\delta_i(\tau_2) = \delta_i(\tau_1\tau_2)$, this is the same as $\delta_i(\tau_1\tau_2) \rightsquigarrow^* \nu_i'$.
- We have what we need and can conclude $\delta_1(\tau_1\tau_2) \approx_{\ell_o} \delta_2(\tau_1\tau_2) : \kappa_2 \sqcup \ell$ by TSCLR:BASE.

Case

$$\frac{\begin{array}{ll}\Delta \vdash \tau : \star^\ell & \Delta \vdash \tau_\to : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\mathbf{bool}} : \kappa & \Delta \vdash \tau_\times : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa\end{array} \quad \text{where } \ell' = \mathcal{L}(\kappa) \text{ and } \ell \sqsubseteq \ell'}{\Delta \vdash \mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times : \kappa}\ \text{WFC:TREC}$$

- By appealing to the induction hypothesis on $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ and
  - $\Delta \vdash \tau : \star^\ell$ and
  - $\Delta \vdash \tau_{\mathbf{bool}} : \kappa$ and
  - $\Delta \vdash \tau_\to : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$ and
  - $\Delta \vdash \tau_\times : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$

  yields
  - $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \star^\ell$ and
  - $\delta_1(\tau_{\mathbf{bool}}) \approx_{\ell_o} \delta_2(\tau_{\mathbf{bool}}) : \kappa$ and
  - $\delta_1(\tau_\to) \approx_{\ell_o} \delta_2(\tau_\to) : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$ and
  - $\delta_1(\tau_\times) \approx_{\ell_o} \delta_2(\tau_\times) : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$

- Using Lemma C·3·17 on these facts gives us that

  $$\mathbf{Typerec}\ \delta_1(\tau)\ \delta_1(\tau_{\mathbf{bool}})\ \delta_1(\tau_\to)\ \delta_1(\tau_\times) \approx_{\ell_o} \mathbf{Typerec}\ \delta_2(\tau)\ \delta_2(\tau_{\mathbf{bool}})\ \delta_2(\tau_\to)\ \delta_2(\tau_\times) : \kappa$$

  By the definition of substitution this is identical to

  $$\delta_1(\mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times) \approx_{\ell_o} \delta_2(\mathbf{Typerec}\ \tau\ \tau_{\mathbf{bool}}\ \tau_\to\ \tau_\times) : \kappa$$

Case

$$\frac{\Delta \vdash \tau : \kappa_1 \qquad \kappa_1 \le \kappa_2}{\Delta \vdash \tau : \kappa_2}\ \text{WFC:SUB}$$

- First, appeal to the induction hypothesis on $\Delta \vdash \tau : \kappa_1$ with $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ to conclude $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \kappa_1$.
- Using Lemma C·3·13 Part 1. on this with $\kappa_1 \sqsubseteq \kappa_2$ we can conclude the desired result, $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \kappa_2$.

Part 2 follows by induction over the structure/heights of typing derivations.

Cases The cases for WFT:TRUE and WFT:FALSE are analogous to that for WFC:BOOL.

Case

$$\frac{\Delta^\star \vdash \Gamma \qquad x : \sigma \in \Gamma}{\Delta^\star; \Gamma \vdash x : \sigma}\ \textsc{wft:var}$$

- Follows immediately by inversion upon $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$.

Cases The cases for WFT:ABS and WFT:APP are analogous to those for WFC:ABS and WFC:APP.

Case

$$\frac{\Delta^\star, \alpha{:}\star^\ell; \Gamma \vdash e : \sigma}{\Delta^\star; \Gamma \vdash \Lambda\alpha{:}\star^\ell.e : \forall^\bot \alpha{:}\star^\ell.\sigma}\ \textsc{wft:tabs}$$

- By the definition of substitution, we know that $\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e)) = \Lambda\alpha{:}\star^\ell.\delta_i(\gamma_i(e))$. Furthermore, by TRC:REFL we know that $\Lambda\alpha{:}\star^\ell.\delta_i(\gamma_i(e)) \rightsquigarrow^* \Lambda\alpha{:}\star^\ell.\delta_i(\gamma_i(e))$. Therefore, we have that $(\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e)) \rightsquigarrow^* (\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e))$.
- Assume $\delta_1(\tau_1) \approx_{\ell_o} \delta_2(\tau_2) : \star^\ell$ and a consistent R such that

$$R^{\ell_2}_\rho \in \delta_1((\rho\{\tau_1\}) \mathbin{@} \ell_2) \leftrightarrow \delta_2((\rho\{\tau_2\}) \mathbin{@} \ell_2).$$

- Therefore, by Definition C·3·7 and RELM:REG we know that $\delta_1,\delta_2 \vdash \eta,\alpha \mapsto R : \Delta^\star,\alpha{:}\star^\ell$ and $\delta_1,[\delta_1(\tau_1)/\alpha] \approx_{\ell_o} \delta_2,[\delta_2(\tau_2)/\alpha] : \Delta^\star,\alpha{:}\star^\ell$.
- Appealing to the induction hypothesis on $\Delta^\star,\alpha{:}\star^\ell; \Gamma \vdash e : \sigma$ with the above gives us that

$$\eta,\alpha \mapsto R \vdash (\delta_1,[\delta_1(\tau_1)/\alpha])(\gamma_1(e)) \approx_{\ell_o} (\delta_2,[\delta_2(\tau_2)/\alpha])(\gamma_2(e)) : \sigma$$

- Using Lemma C·3·10 we can conclude that

$$\eta,\alpha \mapsto R \vdash \delta_1(\gamma_1((\Lambda\alpha{:}\star^\ell.e)[\tau_1])) \approx_{\ell_o} \delta_2(\gamma_2((\Lambda\alpha{:}\star^\ell.e)[\tau_2])) : \sigma$$

  Furthermore, by Lemma C·3·13 and $\Delta^\star \vdash \sigma \le \sigma \sqcup \bot$ we know that

$$\eta,\alpha \mapsto R \vdash \delta_1(\gamma_1((\Lambda\alpha{:}\star^\ell.e)[\tau_1])) \approx_{\ell_o} \delta_2(\gamma_2((\Lambda\alpha{:}\star^\ell.e)[\tau_2])) : \sigma \sqcup \bot$$

- Discharging our assumptions, we have that

$$\eta \vdash \delta_1(\gamma_1(\Lambda\alpha{:}\star^\ell.e)) \sim_{\ell_o} \delta_2(\gamma_2(\Lambda\alpha{:}\star^\ell.e)) : \forall^\bot \alpha{:}\star^\ell.\sigma$$

  Using this along with $(\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e)) \rightsquigarrow^* (\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e))$ and SCLR:TERM we can conclude that

$$\eta \vdash \delta_1(\gamma_1(\Lambda\alpha{:}\star^\ell.e)) \approx_{\ell_o} \delta_2(\gamma_2(\Lambda\alpha{:}\star^\ell.e)) : \forall^\bot \alpha{:}\star^\ell.\sigma$$

Case

$$\frac{\Delta^\star; \Gamma \vdash e : \forall^\ell \alpha{:}\star^{\ell'}.\sigma \qquad \Delta^\star \vdash \tau : \star^{\ell'}}{\Delta^\star; \Gamma \vdash e[\tau] : \sigma[\tau/\alpha] \sqcup \ell}\ \textsc{wft:tapp}$$

- Appealing to the induction hypothesis on $\Delta^\star;\Gamma \vdash e : \forall^\ell \alpha{:}\star^{\ell'}.\sigma$, we get that $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_1} \delta_2(\gamma_2(e)) : \forall^\ell \alpha{:}\star^{\ell'}.\sigma$.
- By inversion on $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_0} \delta_2(\gamma_2(e)) : \forall^\ell \alpha{:}\star^{\ell'}.\sigma$ we know that either $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$ or $\delta_i(\gamma_i(e)) \uparrow$.

  Sub-Case $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$.

  - Also inversion we know that, $\forall^\ell \alpha{:}\star^{\ell'}.\sigma' \rightsquigarrow^* \zeta$ and $\eta \vdash v_1 \sim_{\ell_0} v_2 : \zeta$. By inversion on the weak-head reduction we know that $\zeta = \forall^\ell \alpha{:}\star^{\ell'}.\sigma$. Inverting $\eta \vdash v_1 \sim_{\ell_1} v_2 : \forall^\ell \alpha{:}\star^{\ell'}.\sigma$ we know that

    $$\begin{array}{l} \forall (\delta_1(\tau_1') \approx_{\ell_0} \delta_2(\tau_2') : \star^{\ell'}). \\ \quad \forall (R_\rho^{\ell'} \in \delta_1((\rho\{\tau_1'\}) \mathbin{@} \ell') \leftrightarrow \delta_2((\rho\{\tau_2'\}) \mathbin{@} \ell'). \\ \qquad \eta, \alpha \mapsto R \vdash v_1[\tau_1] \approx_{\ell_1} v_2[\tau_2] : \sigma \sqcup \ell \end{array}$$

  - Using Part 1 on $\Delta^\star \vdash \tau : \star^{\ell'}$ we have that $\delta_1(\tau) \approx_{\ell_0} \delta_2(\tau) : \star^{\ell'}$.
  - Choose $R_\rho^{\ell'}$ to be

    $$\{(v_1, v_2) \;\middle|\; \eta \vdash v_1 \sim_{\ell_0} v_2 : \zeta, (\rho\{\tau\}) \mathbin{@} \ell' \rightsquigarrow^* \zeta\}.$$

  - Applying $\delta_1(\tau) \approx_{\ell_0} \delta_2(\tau) : \star^{\ell'}$ and R gives us that

    $$\eta, \alpha \mapsto R \vdash v_1[\delta_1(\tau)] \approx_{\ell_1} v_2[\delta_2(\tau)] : \sigma \sqcup \ell$$

    Using Lemma C·3·16 on this we can conclude

    $$\eta \vdash v_1[\delta_1(\tau)] \approx_{\ell_1} v_2[\delta_2(\tau)] : \sigma[\tau/\alpha] \sqcup \ell$$

  - Given that $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$ we know that $\delta_i(\gamma_i(e))[\delta_i(\tau)] \rightsquigarrow^* v_i[\delta_i(\tau)]$.
    Using Lemma C·3·10 we can conclude that

    $$\eta \vdash \delta_1(\gamma_1(e))[\delta_1(\tau)] \approx_{\ell_1} \delta_1(\gamma_2(e))[\delta_2(\tau)] : \sigma[\tau/\alpha] \sqcup \ell$$

    which by the definition of substitution is identical to the desired result

    $$\eta \vdash \delta_1(\gamma_1(e[\tau])) \approx_{\ell_1} \delta_1(\gamma_2(e[\tau])) : \sigma[\tau/\alpha] \sqcup \ell$$

  Sub-Case $\delta_i(\gamma_i(e)) \uparrow$.

  - Then we know that $\delta_i(\gamma_i(e[\tau])) \uparrow$ as well. Using SCLR:DIVR1 or SCLR:DIVR2 we can conclude $\eta \vdash \delta_1(\gamma_1(e[\tau])) \approx_{\ell_0} \delta_2(\gamma_2(e[\tau])) : \sigma[\tau/\alpha] \sqcup \ell$.

Case

$$\frac{\Delta^\star;\Gamma \vdash e_1 : \sigma_1 \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma_2}{\Delta^\star;\Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \times^\perp \sigma_2}\ \text{WFT:PAIR}.$$

- By appealing to the induction hypothesis on $\Delta^\star;\Gamma \vdash e_1 : \sigma_1$ and $\Delta^\star;\Gamma \vdash e_2 : \sigma_2$ with $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ and $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ and $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ we have that

$$\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : \sigma_1$$

and

$$\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : \sigma_2$$

- By inversion on $\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : \sigma_1$ either $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_{1i}$ or $\delta_i(\gamma_i(e_1)) \uparrow$.

  Sub-Case $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_{1i}$.

  - By inversion upon $\eta \vdash \delta_1(\gamma_1(e_2)) \approx_{\ell_o} \delta_2(\gamma_2(e_2)) : \sigma_2$ either $\delta_i(\gamma_i(e_2)) \rightsquigarrow^* v_{2i}$ or $\delta_i(\gamma_i(e_2)) \uparrow$.

    Sub-Sub-Case $\delta_i(\gamma_i(e_2)) \rightsquigarrow^* v_{2i}$.

    · Because $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_{1i}$ and $\delta_i(\gamma_i(e_2)) \rightsquigarrow^* v_{2i}$ we can conclude that $\langle\delta_i(\gamma_i(e_1)),\delta_i(\gamma_i(e_2))\rangle \rightsquigarrow^* \langle v_{1i},v_{2i}\rangle$ which by the definition of substitution is identical to $\delta_i(\gamma_i(\langle e_1,e_2\rangle)) \rightsquigarrow^* \langle v_{1i},v_{2i}\rangle$.

    · Therefore, $\mathbf{fst}\,\delta_i(\gamma_i(\langle e_1,e_2\rangle)) \rightsquigarrow^* v_{1i}$ and $\mathbf{snd}\,\delta_i(\gamma_i(\langle e_1,e_2\rangle)) \rightsquigarrow^* v_{2i}$ respectively. Also by the above inversions upon

$$\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : \sigma_1$$

    and

$$\eta \vdash \delta_1(\gamma_1(e_2)) \approx_{\ell_o} \delta_2(\gamma_2(e_2)) : \sigma_2$$

    we know that $\eta \vdash v_{11} \sim_{\ell_o} v_{12} : \zeta_1$ and $\eta \vdash v_{21} \sim_{\ell_o} v_{22} : \zeta_2$ for $\sigma_1 \rightsquigarrow^* \zeta_1$ and $\sigma_2 \rightsquigarrow^* \zeta_2$.

    · Using Lemma C·3·13 on these along with $\Delta^\star \vdash \zeta_i \leq \zeta_i \sqcup \bot$ and $\Delta^\star \vdash \sigma_i \leq \sigma_i \sqcup \bot$ we have that $\eta \vdash v_{11} \sim_{\ell_o} v_{12} : \zeta_1 \sqcup \bot$ and $\eta \vdash v_{21} \sim_{\ell_o} v_{22} : \zeta_2 \sqcup \bot$ for $\sigma_1 \sqcup \bot \rightsquigarrow^* \zeta_1 \sqcup \bot$ and $\sigma_2 \sqcup \bot \rightsquigarrow^* \zeta_2 \sqcup \bot$.

    · Consequently, by SCLR:TERM we have that

$$\eta \vdash \mathbf{fst}\,\delta_1(\gamma_1(\langle e_1,e_2\rangle)) \approx_{\ell_o} \mathbf{fst}\,\delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_1 \sqcup \bot$$

    and

$$\eta \vdash \mathbf{snd}\,\delta_1(\gamma_1(\langle e_1,e_2\rangle)) \approx_{\ell_o} \mathbf{snd}\,\delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_2 \sqcup \bot$$

    · Finally, by SLR:PROD we can conclude

$$\eta \vdash \delta_1(\gamma_1(\langle e_1,e_2\rangle)) \sim_{\ell_o} \delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_1 \times^\bot \sigma_2$$

    Using this along with $\langle\delta_i(\gamma_i(e_1)),\delta_i(\gamma_i(e_2))\rangle \rightsquigarrow^* \langle v_{1i},v_{2i}\rangle$ gives us the desired result

$$\eta \vdash \delta_1(\gamma_1(\langle e_1,e_2\rangle)) \approx_{\ell_o} \delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_1 \times^\bot \sigma_2$$

    Sub-Sub-Case $\delta_i(\gamma_i(e_2)) \uparrow$.

· Then we know that $\delta_i(\gamma_i(\langle e_1, e_2\rangle)) \uparrow$ and we can use either SCLR:DIVR1 or SCLR:DIVR2 to conclude that

$$\eta \vdash \delta_1(\gamma_1(\langle e_1, e_2\rangle)) \approx_{\ell_o} \delta_2(\gamma_2(\langle e_1, e_2\rangle)) : \sigma_1 \times^{\perp} \sigma_2$$

Sub-Case $\delta_i(\gamma_i(e_1)) \uparrow$.

- Then we know that $\delta_i(\gamma_i(\langle e_1, e_2\rangle)) \uparrow$ and we can use either SCLR:DIVR1 or SCLR:DIVR2 to conclude that

$$\eta \vdash \delta_1(\gamma_1(\langle e_1, e_2\rangle)) \approx_{\ell_o} \delta_2(\gamma_2(\langle e_1, e_2\rangle)) : \sigma_1 \times^{\perp} \sigma_2$$

Case

$$\frac{\Delta^\star; \Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2}{\Delta^\star; \Gamma \vdash \mathbf{fst}\, e : \sigma_1 \sqcup \ell} \text{ WFT:FST}$$

- Appealing to the induction hypothesis on $\Delta^\star; \Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2$ we know that $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma_1 \times^{\ell} \sigma_2$.
- By inversion upon $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma_1 \times^{\ell} \sigma_2$ we know that either $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$ or $\delta_i(\gamma_i(e)) \uparrow$.

  Sub-Case $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$,

  - Also by inversion upon

$$\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma_1 \times^{\ell} \sigma_2$$

    we have that $\sigma_1 \times^{\ell} \sigma_2 \rightsquigarrow^* \sigma' \; \eta \vdash v_1 \sim_{\ell_o} v_2 : \sigma'$.
  - By inversion upon $\sigma_1 \times^{\ell} \sigma_2 \rightsquigarrow^* \sigma'$ we know that $\sigma' = \sigma_1 \times^{\ell} \sigma_2$.
  - By inversion upon $\eta \vdash v_1 \sim_{\ell_o} v_2 : \sigma_1 \times^{\ell} \sigma_2$ we know that $\eta \vdash \mathbf{fst}\, v_1 \approx_{\ell_o} \mathbf{fst}\, v_2 : \sigma_1 \sqcup \ell$ and $\eta \vdash \mathbf{snd}\, v_1 \approx_{\ell_o} \mathbf{snd}\, v_2 : \sigma_2 \sqcup \ell$.
  - Given that $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$ we know that $\mathbf{fst}\, \delta_i(\gamma_i(e)) \rightsquigarrow^* \mathbf{fst}\, v_i$ which by the definition of substitution is the same as $\delta_i(\gamma_i(\mathbf{fst}\, e)) \rightsquigarrow^* \mathbf{fst}\, v_i$. Therefore by Lemma C·3·10 we can conclude that

$$\eta \vdash \delta_1(\gamma_1(\mathbf{fst}\, e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fst}\, e)) : \sigma_1 \sqcup \ell$$

  Sub-Case $\delta_i(\gamma_i(e)) \uparrow$

  - Therefore, we can conclude that $\mathbf{fst}\, \delta_i(\gamma_i(e)) \uparrow$, which by the definition of substitution is the same as $\delta_i(\gamma_i(\mathbf{fst}\, e)) \uparrow$. Therefore, by SCLR:DIVR1 or SCLR:DIVR2 we have that $\eta \vdash \delta_1(\gamma_1(\mathbf{fst}\, e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fst}\, e)) : \sigma_1 \sqcup \ell$.

Case The case for WFT:SND is symmetric to the case for WFT:FST.

Case

$$\frac{\Delta^\star; \Gamma \vdash e_1 : (\mathbf{bool})\ @\ \ell \qquad \Delta^\star; \Gamma \vdash e_2 : \sigma \qquad \Delta^\star; \Gamma \vdash e_3 : \sigma}{\Delta^\star; \Gamma \vdash \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 : \sigma \sqcup \ell} \text{ WFT:IF}$$

Sub-Case $\ell \not\sqsubseteq \ell_o$.

- Then by Lemma C·3·15 Part 2 we know that

$$\eta \vdash \delta_1(g_1(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)) \approx_{\ell_o} \delta_2(g_2(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)) : \sigma \sqcup \ell$$

Sub-Case $\ell \sqsubseteq \ell_o$.

- By appealing to the induction hypothesis on $\Delta^\star; \Gamma \vdash e_1 : (\textbf{bool}) \mathbin{@} \ell$ we know that $\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : (\textbf{bool}) \mathbin{@} \ell$. By inversion on this we know that either $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_i$ or $\delta_i(\gamma_i(e_1)) \uparrow$.

  Sub-Sub-Case $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_i$.

  - Also by inversion we know that $\eta \vdash v_1 \sim_{\ell_o} v_2 : \zeta$ where $(\textbf{bool}) \mathbin{@} \ell \rightsquigarrow^* \zeta$. And by inversion on the weak-head reduction we know that $\zeta = (\textbf{bool}) \mathbin{@} \ell$.
  - Therefore, by inversion upon $\eta \vdash v_1 \sim_{\ell_o} v_2 : (\textbf{bool}) \mathbin{@} \ell$ we can conclude $\ell \sqsubseteq \ell_o \implies v_1 = v_2$. We assumed that $\ell \sqsubseteq \ell_o$, so $v_1 = v_2$.
  - By Lemma C·1·15 we know that $v_i = \textbf{true}$ or $v_i = \textbf{false}$.

    Sub-Sub-Sub-Case $v_i = \textbf{true}$. By appealing to the induction hypothesis on $\Delta^\star; \Gamma \vdash e_1 : (\textbf{bool}) \mathbin{@} \ell$ we know that

$$\eta \vdash \delta_1(\gamma_1(e_2)) \approx_{\ell_o} \delta_2(\gamma_2(e_2)) : \sigma$$

    By Lemma C·3·13 we can conclude

$$\eta \vdash \delta_1(\gamma_1(e_2)) \approx_{\ell_o} \delta_2(\gamma_2(e_2)) : \sigma \sqcup \ell$$

    We know that $\delta_i(\gamma_i(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)) \rightsquigarrow^* \delta_i(\gamma_i(e_2))$, therefore by Lemma C·3·10 we can conclude the desired result

$$\eta \vdash \delta_1(g_1(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)) \approx_{\ell_o} \delta_2(g_2(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)) : \sigma \sqcup \ell$$

    Sub-Sub-Sub-Case The case for $v_i = \textbf{false}$ is symmetric.

  Sub-Sub-Case $\delta_i(\gamma_i(e_1)) \uparrow$.

  - Then we know that $\delta_i(g_i(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)) \uparrow$ and can use either SCLR:DIVR1 or SCLR:DIVR2 to conclude that

$$\eta \vdash \delta_1(g_1(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)) \approx_{\ell_o} \delta_2(g_2(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)) : \sigma \sqcup \ell$$

Case

$$\frac{\Delta^\star; \Gamma, x{:}\sigma \vdash e : \sigma \qquad \Delta^\star \vdash \sigma}{\Delta^\star; \Gamma \vdash \textbf{fix}_n\ x{:}\sigma.e : \sigma}\ \textsc{wft:fixn}$$

- By the definition of substitution, we know that $\delta_i(\gamma_i(\textbf{fix}_n\ x{:}\sigma.e)) = \textbf{fix}_n\ x{:}\sigma.\delta_i(\gamma_i(e))$.
- The case follows from induction upon n.

  Sub-Case $n = 0$.

- By Lemma C·2·5 we know that $\mathbf{fix}_o\ x{:}\sigma.\delta_i(\gamma_i(e))\uparrow$. Therefore, by SCLR:DIVR1 or SCLR:DIVR20 we can conclude that

$$\eta \vdash \mathbf{fix}_o\ x{:}\sigma.\delta_1(\gamma_1(e)) \approx_{\ell_o} \mathbf{fix}_o\ x{:}\sigma.\delta_2(\gamma_2(e)) : \sigma$$

- By the above identity, this means that we have

$$\eta \vdash \delta_1(\gamma_1(\mathbf{fix}_o\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}_o\ x{:}\sigma.e)) : \sigma$$

Sub-Case $n = m + 1$.

- By appealing to the local induction hypothesis on m gives us that $\eta \vdash \delta_1(\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)) : \sigma$.
- By Definition C·3·9 and inversion upon $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ we can conclude that

$$\eta \vdash \gamma_1,[\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)/x] \approx_{\ell_o} \gamma_2,[\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)/x] : \Gamma, x{:}\sigma$$

- Appealing to the global induction hypothesis on $\Delta^\star;\Gamma,x{:}\sigma \vdash e : \sigma$ with

$$\eta \vdash \gamma_1,[\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)/x] \approx_{\ell_o} \gamma_2,[\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)/x] : \Gamma, x{:}\sigma$$

gives us that

$$\eta \vdash \delta_1((\gamma_1,[\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)/x])(e)) \approx_{\ell_o} \delta_2((\gamma_2,[\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)/x])(e)) : \sigma$$

- Trivially, $n - 1 = m$, so using Lemmas C·3·10 on

$$\eta \vdash \delta_1((\gamma_1,[\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)/x])(e)) \approx_{\ell_o} \delta_2((\gamma_2,[\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)/x])(e)) : \sigma$$

we can conclude

$$\eta \vdash \delta_1(\gamma_1(\mathbf{fix}_n\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}_n\ x{:}\sigma.e)) : \sigma$$

Case

$$\frac{\Delta^\star;\Gamma,x{:}\sigma \vdash e : \sigma \qquad \Delta^\star \vdash \sigma}{\Delta^\star;\Gamma \vdash \mathbf{fix}\ x{:}\sigma.e : \sigma}\ \text{WFT:FIX}$$

- Using Lemma C·2·6 we know that for all$n$, $\Delta^\star;\Gamma \vdash \mathbf{fix}_n\ x{:}\sigma.e : \sigma$.
- Therefore, assume an arbitrary m. Appealing to the induction hypothesis on $\Delta^\star;\Gamma \vdash \mathbf{fix}_m\ x{:}\sigma.e : \sigma$ with $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ gives us that $\eta \vdash \delta_1(\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)) : \sigma$.
- By the definition of substitution $\delta_i(\gamma_i(\mathbf{fix}_m\ x{:}\sigma.e)) = \mathbf{fix}_m\ x{:}\delta_i(\sigma).\delta_i(\gamma_i(e))$. Therefore, we have that

$$\eta \vdash \mathbf{fix}_m\ x{:}\delta_1(\sigma).\delta_1(\gamma_1(e)) \approx_{\ell_o} \mathbf{fix}_m\ x{:}\delta_2(\sigma).\delta_2(\gamma_2(e)) : \sigma$$

- Discharging our assumption we have that for all $\mathfrak{n}$,

$$\eta \vdash \mathbf{fix}_{\mathfrak{n}}\ x{:}\delta_1(\sigma).\delta_1(\gamma_1(e)) \approx_{\ell_o} \mathbf{fix}_{\mathfrak{n}}\ x{:}\delta_2(\sigma).\delta_2(\gamma_2(e)) : \sigma$$

  Using Lemma 2·2·5 we can conclude

$$\eta \vdash \mathbf{fix}\ x{:}\delta_1(\sigma).\delta_1(\gamma_1(e)) \approx_{\ell_o} \mathbf{fix}\ x{:}\delta_2(\sigma).\delta_2(\gamma_2(e)) : \sigma$$

- Again by the definition of substitution, $\delta_i(\gamma_i(\mathbf{fix}\ x{:}\sigma.e)) = \mathbf{fix}\ x{:}\delta_i(\sigma).\delta_i(\gamma_i(e))$. Therefore, we have the desired result

$$\eta \vdash \delta_1(\gamma_1(\mathbf{fix}\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}\ x{:}\sigma.e)) : \sigma$$

Case The case for WFT:TCASE is analogous to WFT:IF and WFT:TAPP.

Case The case for WFT:SUB is analogous to that for WFC:SUB.

□

Corollary C·3·20 (Confidentiality)**.** *If* $\alpha{:}\star^\top; x{:}(\alpha)\ @\ \bot \vdash e : (\mathbf{bool})\ @\ \bot$ *then for any* $\cdot \vdash v_1 : \tau_1$ *and* $\vdash v_2 : \tau_2$ *if* $e[\tau_1/\alpha][v_2/x]$ *and* $e[\tau_2/\alpha][v_2/x]$ *both terminate, they will produce the same value.*

Proof. Then construct a derivation that $\cdot;\cdot \vdash \Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e : \forall\alpha{:}\star^\top.(\alpha)\ @\ \bot \xrightarrow{\bot} (\mathbf{bool})\ @\ \bot$ using the appropriate typing rules and then appeal to Theorem C·3·19 Part 2 to obtain

$$\cdot \vdash \Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e \sim_\bot \Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e : \forall\alpha{:}\star^\top.(\alpha)\ @\ \bot \xrightarrow{\bot} (\mathbf{bool})\ @\ \bot$$

By Lemma C·3·15 Part 1 we can have that $\tau_1 \approx_\bot \tau_2 : \star^\top$. Next, by inversion on SLR:ALL and instantiation with the constructor relation, $\tau_1 \approx_\bot \tau_2 : \star^\top$, and the relation

$$R^\ell_\rho = \{(v_1, v_2) \mid (\cdot;\cdot \vdash v_1 : (\rho\{\tau_1\})\ @\ \ell), (\cdot;\cdot \vdash v_2 : (\rho\{\tau_2\})\ @\ \ell)\},$$

we can conclude that

$$\cdot, \alpha \mapsto R \vdash (\Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e)[\tau_1] \approx_\bot (\Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e)[\tau_2] : (\alpha)\ @\ \bot \xrightarrow{\bot} (\mathbf{bool})\ @\ \bot$$

By straightforward application of SLR:VAR we have that

$$\cdot, \alpha \mapsto R \vdash v_1 \sim_\bot v_2 : (\alpha)\ @\ \bot$$

so by application of SCLR:TERM, inversion on SCLR:ARR, and instantiation we know

$$\cdot, \alpha \mapsto R \vdash (\Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e)[\tau_1]v_1 \approx_\bot (\Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e)[\tau_2]v_2 : (\mathbf{bool})\ @\ \bot$$

Finally, because the relation is closed under reduction we have SLR:ARR and instantiation we have

$$\cdot, \alpha \mapsto R \vdash e[\tau_1/\alpha][v_1/x] \approx_\bot e[\tau_2/\alpha][v_2/x] : (\mathbf{bool})\ @\ \bot$$

from which the desired conclusion can be obtained by simple inversion. □

Corollary C·3·21 (Noninterference)**.** *If* $\cdot,x{:}\sigma_1 \vdash e : \sigma_2$ *where* $\mathcal{L}(\sigma_1) \not\sqsupseteq \mathcal{L}(\sigma_2)$ *then for any* $\vdash v_1 : \sigma_1$ *and* $\vdash v_2 : \sigma_1$ *it is the case that if both* $e[v_1/x]$ *and* $e[v_2/x]$ *terminate, they will both produce the same value*

Proof. Proceeds in a similar fashion to Corollary C·3·20. □

Corollary C·3·22 (Integrity)**.** *If* $\alpha{:}\star^\top; \cdot \vdash e : (\alpha) \mathbin{@} \bot$ *then* $e[\tau/\alpha]$ *for any* τ *must diverge.*

Proof. First construct a derivation that $\cdot; \cdot \vdash \Lambda\alpha{:}\star^\top.e : \forall\alpha^\top{:}(\alpha) \mathbin{@} \bot$ using the appropriate typing rules, then appeal to Theorem C·3·19 Part 2 to obtain to obtain

$$\cdot \vdash \Lambda\alpha{:}\star^\top.e \sim_\bot \Lambda\alpha{:}\star^\top.e : \forall\alpha{:}\star^\top.(\alpha) \mathbin{@} \bot$$

Now assume an arbitrary τ. It is straightforward to show that $\tau \approx_\bot \tau : \star^\top$. By inversion on SLR:ALL and instantiation we can conclude

$$\cdot,\alpha \mapsto \varnothing \vdash (\Lambda\alpha{:}\star^\top.e)[\tau] \approx_\bot (\Lambda\alpha{:}\star^\top.e)[\tau] : (\alpha) \mathbin{@} \bot$$

Because the relation is closed under reduction we have that

$$\cdot,\alpha \mapsto \varnothing \vdash e[\tau/\alpha] \approx_\bot e[\tau/\alpha] : (\alpha) \mathbin{@} \bot$$

Furthermore, by inversion either $e[\tau/\alpha] \rightsquigarrow^* v$ or $e[\tau/\alpha]\uparrow$. However in the former case that would mean that

$$\cdot,\alpha \mapsto \varnothing \vdash v \sim_\bot v : (\alpha) \mathbin{@} \bot$$

which by inversion on SLR:VAR is impossible because there is no v such that $v \varnothing v$. Therefore $e[\tau/\alpha]\uparrow$. □

D

Full grammar of the InforML language

§ D·1 Identifiers and other miscellany

Note that several Roman and Greek glyphs look identical, like Roman `A` (Unicode 0041) and Greek `A` (Unicode 0391), but are treated as distinct glyphs. The POSIX style regular expressions (Corporate IEEE Staff 1993) below specify the acceptable lexical forms for identifiers

- Symbol are `[!%&+/:<=>?@~*-^|]`*.
- Variable identifiers are symbols or `[a-zα-ω][A-ZA-Ωa-zα-ω_0-9]`*.
- Constructor identifiers are symbols or `[A-ZA-Ω][A-ZA-Ωa-zα-ω_0-9]`*. Constructor identifiers are not α-convertible.
- Record selectors and atoms can be either variable identifiers or constructor identifiers. Record selectors and atoms are not α-convertible.

The above specification has the following exceptions:

- `Π` may not be used a constructor identifier.
- `λ` may not be used a variable identifier.
- The symbols `*` and `%`, may be used as variable identifiers, but nothing else.
- The symbols `:`, `|`, `||`, `&&`, `@`, `<:`, `:>`, `=`, `=(`, `)=>`, `=>`, `-(`, and `)->`, are reserved for use by the language.

The following table summarizes the meta-variable conventions for identifiers:

Meta-variable	Lexical category	Semantic category
`l`	variable	label variables
`α,β`	variable	type variables
`A`	constructor	algebraic data types
`x,y`	variable	term variables
`D`	constructor	data constructors
`m`	variable	module variables
`s`	variable	signature variables
`m`	variable	module variables
`r`	either	record selectors
`a`	either	atoms

Numbers

I use the meta-variable `n` for natural numbers `0`, `1`, ... and the meta-variable `i` for the integers ..., `-1`, `0`, `1`, ...

Comments

Region comments, comments for a potentially multiline region of code, are started by `#(` and ended by `#)`. They may be arbitrarily nested. Line comments, commenting the remainder of a given line, can be started with character sequence `#`. Line comments may be nested within region comments, but at present region comments will not be recognized as starting in a line comment. In fact, this can be useful.

You can comment out a region of code as follows

```
#(
val foo = 1 + 1
#)
```

And by simply adding an additional hash, it is possible to enable the region

```
##(
val foo = 1 + 1
#)
```

§ D·2 The type system

In InforML, types proper are divided into three categories: *polytypes* `σ`, *ρ-types* `ρ`, and *monotypes* `τ`. Types are classified by *kinds*.

Labels

Atomic labels

ℓ	::=	`l`	label variables
	|	`+{(m.)*a}`	additive boolean element
	|	`-{(m.)*a}`	subtractive boolean element

It is also possible to write `Top` or $\top$ for `-{}` and `Bot` or $\bot$ for `+{}`. Using $\top$ or $\bot$ reuqires the input stream be UTF8 encoded.

Full labels

Л	::=	ℓ	atomic labels
	|	`info τ`	information content of a type variable
	|	$Л_1$`||` … `||` $Л_n$	join (for $n > 1$)
	|	$Л_1$`&&` … `&&` $Л_n$	meet (for $n > 1$)
	|	`(Л)`	

$Л_1$`||` … `||` $Л_n$ and $Л_1$`&&` … `&&` $Л_n$ may be written as $Л_1 \sqcup \ldots \sqcup Л_n$ and $Л_1 \sqcap \ldots \sqcap Л_n$, respectively. These alternatives require that the input stream be UTF8 encoded.

Variances

Variances are used in kinds and types to specify the behavior of subtyping.

î	::=	`+`	covariant
	|	`-`	contravariant
	|	`=`	invariant

The variance = may also be written as ±. This alternative requires that the input stream be UTF8 encoded.

Kinds

κ	::=	`* @ Л`	type classifiers
	|	`% @ Л`	algebraic type classifiers
	|	`Lab -(π)->` κ	label functions
	|	`Pi l (:Lab)?-(π)->` κ	dependent label functions
	|	κ_1 `-(π)->` κ_2	type functions

`Pi l (:Lab)? -(π)->` κ may also be written as `Π l (:Lab)? -(π)->` κ. This alternative requires that the input stream be UTF8 encoded.

Constraints

C	::=	$Л_1$ `<:` $Л_2$	
	|	$Л_1$ `:>` $Л_2$	
	|	$Л_1$ `=` $Л_2$	
	|	C_1 `&` … `&` C_n	conjunction (for $n > 1$)

Quantifier blocks

```
qfb  ::=  ⟨(l(:Lab)?)+(| C)?⟩
       |  ⟨(α: κ)+(| C)?⟩
       |  ⟨(l(:Lab)?)+|(α: κ)+(| C)?⟩
```

Polytypes

```
σ  ::=  ∀qfb σ      universal types
     |  ∃qfb σ      existential types
     |  ρ           ρ-types
     |  (σ)
```

ρ-types

```
ρ  ::=  σ1 -(Л1|Л2)-> σ2                  higher-rank term functions
                                          (Л1 for program counter, Л2 for the closure)
     |  (|σ1, ... , σn|)-(Л1|Л2)-> σ       curried higher-rank term functions
                                          (Л1 for program counter, Л2 for the closure)
     |  τ                                 monotypes
     |  (ρ)
```

Monotypes

```
τ  ::=  (m.)*α                            type variables
     |  (m.)*A                            algebraic data types
     |  Int                               primitive integer type
     |  String                            primitive string type
     |  Bool                              primitive boolean type
     |  τ1τ2                              type application
     |  fn (l:Lab | α:κ)=(π)=> τ           type functions
     |  τ @ Л                             label application
     |  {r1:τ1, ... , rn:τn}              record types
                                          (for n ≥ 0 and r1 ... rn distinct)
     |  (τ1, ... , τ_n \verb)|            tuple types (for n > 1)
     |  τ1 -(Л1|Л2)-> τ2                  term functions
                                          (Л1 for program counter, Л2 for the closure)
     |  τ : κ                             kind annotation
     |  (τ)
```

`fn (l:Lab | α:κ)=(π)=> τ` may also be written as `λ (l:Lab | α:κ)=(π)=> τ`. These alternatives requires that the input stream be UTF8 encoded.

§ D·3 Patterns

Label patterns

`lp`	::=	`_`	wildcard label pattern
	|	ℓ	atomic labels

Type patterns

φ	::=	`_`	wildcard type pattern
	|	$(\texttt{m.})^*\alpha$	type variables
	|	$(\texttt{m.})^*\texttt{A}$	algebraic data types
	|	`Int`	primitive integer type pattern
	|	`Bool`	primitive boolean type pattern
	|	`String`	primitive string type pattern
	|	$\varphi_1\ \varphi_2$	type application pattern
	|	φ `@ lp`	label application pattern
	|	φ_1 `-(`$\texttt{lp}_1$`|`$\texttt{lp}_2$`)->` φ_2	term function patterns ($\texttt{lp}_1$ for program counter, $\texttt{lp}_2$ for the closure)
	|	φ_1 `-(`$\texttt{lp}_1$`|`$\texttt{lp}_2$`|`$\texttt{lp}_3$`|`$\texttt{lp}_4$`)->` φ_2	term function patterns ($\texttt{lp}_1$ for information content of φ_1, $\texttt{lp}_2$ for program counter, $\texttt{lp}_3$ for the closure, $\texttt{lp}_4$ for information content of φ_2)
	|	$\texttt{\{}r_1\texttt{:}\varphi_1\texttt{, … ,}r_n\texttt{:}\varphi_n\texttt{\}}$	record type patterns (for $n \geq 0$ and $r_1 \ldots r_n$ distinct)
	|	$(\varphi_1\texttt{, … ,}\ \varphi_n)$	tuple type patterns (for $n > 1$)
	|	φ `:` κ	annotated type pattern
	|	(φ)	

Term patterns

`p`	::=	`_`	wildcard pattern
	|	`x`	variable binding
	|	`i`	integer patterns
	|	(`True` | `False`)	boolean patterns
	|	`"strings"`	string patterns
	|	$(\texttt{m.})^*\texttt{D}\ (\langle \texttt{l}^* \mid \alpha^* \rangle)^?\ \texttt{p}_1\ \ldots\ \texttt{p}_n$	data constructor patterns – must be fully applied
	|	$\texttt{\{}r_1\texttt{=}p_1\texttt{, … ,}r_n\texttt{=}p_n\texttt{\}}$	record patterns (for $n \geq 0$ and $r_1 \ldots r_n$ distinct)
	|	$(\texttt{p}_1\texttt{, … ,}\ \texttt{p}_n)$	tuple patterns (for $n > 1$)
	|	$\texttt{[p}_1\texttt{, … ,}\ \texttt{p}_n\texttt{]}$	list patterns (for $n \geq 0$)
	|	`p :` σ	annotated pattern
	|	`(p)`	

§ D·4 Expressions

Matches

Term matches

```
u  ::=  p => e
     |  p =(ℓ)=> e
```

Type matches

```
μ  ::=  φ => e
```

Expressions proper

`e` ::=	`(m.)*(x | D)(⟨Π* | τ*⟩)?`	instantiation
|	`i`	integers
|	`(True | False)`	booleans
|	`"strings"`	strings
|	`op (x | D)`	forced `nofix`
|	`[e₁, … , eₙ]`	lists (for $n \geq 0$)
|	`{r₁=e₁, … ,rₙ=eₙ}`	records (for $n \geq 0$ and $r_1 \ldots r_n$ distinct)
|	`(e₁, … , eₙ)`	tuples (for $n > 1$)
|	`e.n`	tuple projection
|	`e₁ andalso e₂`	short-circuiting “and”
|	`e₁ orelse e₂`	short-circuiting “or”
|	`fn (|)? u₁ | … | uₙ end`	anonymous functions (for $n > 0$)
|	`let ld* in e end`	let expression
|	`if e₁ then e₂ else e₃ end`	conditional
|	`case e (of | |)u₁ | … | uₙ end`	term case (for $n > 0$)
|	`typecase τ (of | |) μ₁ | … | μₙ end`	type case (for $n > 0$)
|	`ifholds C then e₁ else e₂ end`	dynamic constraint check
|	`isdata α then e₁ else e₂ end`	type constructor cast
|	`e₁ e₂`	term application
|	`e : σ`	type annotation
|	`(e)`	

`fn (|)? u₁ | … | uₙ end` may also be written as `λ (|)? u₁ | … | uₙ end`. This alternative requires that the input stream be UTF8 encoded.

Values

v	::=	$(\texttt{m.})^*(\texttt{x} \mid \texttt{D})\,(\langle \Pi^* \mid \tau^* \rangle)^?$	instantiation
	|	$\texttt{i}$	integers
	|	$(\texttt{True} \mid \texttt{False})$	booleans
	|	$\texttt{"strings"}$	strings
	|	$\texttt{op}\,(\texttt{x} \mid \texttt{D})$	forced `nofix`
	|	$\texttt{[}v_1\texttt{, ... , }v_n\texttt{]}$	lists (for $n \geq 0$)
	|	$\texttt{\{}r_1\texttt{=}v_1\texttt{, ... ,}r_n\texttt{=}v_n\texttt{\}}$	records (for $n \geq 0$ and $r_1 \ldots r_n$ distinct)
	|	$\texttt{(}v_1\texttt{, ... , }v_n\texttt{)}$	tuples (for $n > 1$)
	|	$\texttt{fn}\ (\texttt{\|})^?\ u_1\ \texttt{\|}\ \ldots\ \texttt{\|}\ u_n\ \texttt{end}$	anonymous functions (for $n > 0$)
	|	$\texttt{v : }\sigma$	type annotation
	|	$\texttt{(v)}$	

$\texttt{fn}\ (\texttt{|})^?\ u_1\ \texttt{|}\ \ldots\ \texttt{|}\ u_n\ \texttt{end}$ may also be written as $\lambda\ (\texttt{|})^?\ u_1\ \texttt{|}\ \ldots\ \texttt{|}\ u_n\ \texttt{end}$. This alternative requires that the input stream be UTF8 encoded.

§ D·5 Declarations

Fixity annotations `n` is a precedence somewhere between `0` and `9999`. Note that `prefix` or `postfix` implies that the items in question are unary operators, but this is not enforced by the typechecker.

`fix`	::=	$\texttt{infixr}\,(\texttt{n})^?$	
	|	$\texttt{infixl}\,(\texttt{n})^?$	
	|	$\texttt{prefix}\,(\texttt{n})^?$	
	|	$\texttt{postfix}\,(\texttt{n})^?$	
	|	`nofix`	

Data type binding

`dtb`	::=	$\texttt{A : }\kappa\texttt{ = }(\texttt{\|})^?\ \texttt{D}_1\texttt{:}\sigma_1\ \texttt{\|}\ \ldots\ \texttt{\|}\ \texttt{D}_n\texttt{:}\sigma_n$	algebraic data type binding (for $n \geq 0$)

The head constructor of $\sigma_1 \ldots \sigma_n$ must be `A`.

Recursive function binding

`fb`	::=	$\texttt{x}\,(\langle \texttt{l}^* \mid \alpha^* \rangle)^?\,(\texttt{p})^+\,(\texttt{: }\sigma)^?\texttt{ = e}$	recursive function binding

Local declarations

`ld`	::=	$\texttt{fun fb}_1\ \texttt{and}\ \ldots\ \texttt{and fb}_n$	recursive function definitions (for $n > 0$)
	|	`fun x : σ`	type annotation declaration
	|	`val p = e`	"let" declaration
	|	`do e`	sugar for effectful expressions
	|	$\texttt{fix}(\alpha \mid \texttt{A} \mid \texttt{x} \mid \texttt{D})$	fixity declaration

Declarations proper

`d`	::=	`ld`	local declaration
	|	`newatoms (a)`var^{+}	atoms
	|	`module m (:S)`$^{?}$ `= M`	module declaration
	|	`signature s = S`	signature declaration
	|	`datatype dtb`$_1$ `and ... and dtb`$_n$	data type definitions (for $n > 0$)
	|	`type` α_1 `(: κ`$_1$`)`$^{?}$ `= τ`$_1$ `and ... and` α_n `(: κ`$_n$`)`$^{?}$ `= τ`$_n$	type definitions

§ D·6 Modules and signatures

`M`	::=	`(m.)`*`m`	module variable
	|	`mod d`* `end`	

Signature bindings

`sb`	::=	`atom a`	atom
	|	`data A : κ`	algebraic data type
	|	`type α : κ`	opaque type definition
	|	`type α : κ = τ`	translucent type definition
	|	`con D : σ`	data constructor
	|	`val x : σ`	value
	|	`fun x : σ`	function
	|	`mod m : S`	module

Signatures proper

`S`	::=	`s`	signature variables
	|	`sig sb`*`end`	

Bibliography

Abadi, Martín, Anindya Banerjee, Nevin Heintze, and Jon Riecke. 1999.
A core calculus of dependency.
In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 147–160. San Antonio, TX.

Abadi, Martín, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. 1990.
Explicit substitutions.
In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 31–46. San Francisco, CA:Acm Press.

Abadi, Martín, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991.
Dynamic typing in a statically typed language.
ACM Transactions on Programming Languages and Systems 13(2):237–268.
Also appeared as Src Research Report 47.

Achten, Peter, Marko van Eekelen, Rinus Plasmeijer, and Arjen van Weelden. 2004a.
Automatic generation of editors for higher-order data structures.
Technical Report Niii–r0427, Radboud University Nijmegen.

Achten, Peter, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. 2004b.
Compositional model-views with generic graphical user interfaces.
In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, volume 3057 of *Lecture Notes in Computer Science*, 39–55. Dallas, TX:Springer-Verlag.

Ahmed, Amal J. 2004.
Semantics of types for mutable state.
Ph.D. thesis, Princeton University.
Technical Report Tr-713-04.

———. 2006.
Step-indexed syntactic logical relations for recursive and quantified types.
In *15th European Symposium on Programming*, edited by ?, volume 3924 of *Lecture Notes in Computer Science*, 69–83. Vienna, Austria:Springer-Verlag.

Appel, Andrew W., and David McAllester. 2001.
An indexed model of recursive types for foundational proof-carrying code.
ACM Transactions on Programming Languages and Systems 23(5):657–683.

Aspinall, David, and Adriana Compagnoni. 2001.
Subtyping dependent types.
Theoretical Computer Science 266(1-2):273–309.

Aydemir, Brian, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008.
Engineering formal metatheory.
In *Proceedings of the 35th ACM SIGPLAN - SIGACT symposium on Principles of programming languages*. San Fransico, CA:Acm Press.
To appear.

Aydemir, Brian E., Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005.
Mechanized metatheory for the masses: The POPLmark challenge.
In *18th International Conference on Theorem Proving in Higher Order Logics*, edited by Joe Hurd and Tom Melham, volume 3603 of *Lecture Notes in Computer Science*, 50–65. Oxford, UK.

Bauer, Lujo, Jarred Ligatti, and David Walker. 2004.
A language and system for composing security policies.
Technical Report Tr-699-04, Princeton University.

Bell, David E., and Leonard J. La Padula. 1975.
Secure computer system: Unified exposition and Multics interpretation.
Technical Report Esd-tr-75-306, Mitre Corporation Mtr-2997, Bedford, MA.

Biba, Kenneht J. 1977.
Integrity considerations for secure computer systems.
Technical Report Esd-tr-76-372, Usaf Electronic Systems Division, Bedford, MA.

Birkedal, Lars, and Robert Harper. 1999.
Relational interpretations of recursive types in an operational setting.
Information and Computation 155:3–63.

Blume, Matthias. 2007.
The SML/NJ Compilation and Library Manager User Manaul.
For sml/nj version 110.65.

Breazu-Tannen, Val, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. 1991.
Inheritance as implicit coercion.
Information and Computation 93:172–221.

Brus, Tom H., Marko C. J. D. van Eekelen, M.O van Leer, and Marinus J. Plasmeijer. 1987.
Clean: A language for functional graph rewriting.

In *Proc. of a conference on functional programming languages and computer architecture*, volume 274 of *Lecture Notes in Computer Science*, 364–384. Portland, OR:Springer-Verlag.

Chen, Gang. 2003.
Coercive subtyping for the calculus of constructions.
In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 150–159. New Orleans, LA:Acm Press.

Cheney, James. 2005.
Scrap your nameplate (functional pearl).
In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, 180–191. Tallinn, Estonia:Acm Press.

Chong, Stephen, Andrew C. Myers, K. Vikram, and Lantian Zheng.
Jif reference manual.
Located at `http://www.cs.cornell.edu/jif/`.

Consortium, The Unicode. 2006.
The Unicode Standard 5.0.
5th edition. Redwood City, CA:Addison-Wesley.

Coquand, Thierry. 1992.
Pattern matching with dependent types.
In *Informal proceedings workshop on types for proofs and programs*, edited by Bengt Nordström, Kent Pettersson, and Gordan Plotkin, 71–84. Båstad, Sweden:Deptmart of Computing Science, Chalmers University of Technology and Göteborg University.

Corporate Ieee Staff. 1993.
ISO-IEC 9945-2: IEEE Standard 1003.2-1992 Information Technology - Portable Operating System Interface: Shell and Utilities.
New York, NY, USA:Ieee Standards Office.

Crary, Karl, and Robert Harper. 2007.
Syntactic logical relations for polymorphic and recursive types.
Electronic Notes Theoretical Computer Science 172:259–299.

Crary, Karl, and Stephanie Weirich. 1999.
Flexible type analysis.
In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, 233–248. Paris, France:Acm Press.

Crary, Karl, Stephanie Weirich, and Greg Morrisett. 2002.
Intensional polymorphism in type erasure semantics.
The Journal of Functional Programming 12(6):567–600.

Damas, Luis, and Robin Milner. 1982.
Principal type schemes for functional programs.
In *Proceedings of the 9th acm sigplan-sigact symposium on principles of programming languages*, 207–212. Albuquerque, NM:Acm Press.

Dantas, Daniel S. 2007.
Harmless advice.
Ph.D. thesis, Princeton University.
Technical Report Tr-795-07.

Dantas, Daniel S., David Walker, Geoffrey Washburn, and Stephanie Weirich. 2008.
AspectML: A polymorphic aspect-oriented functional programming language.
ACM Transactions on Programming Languages and Systems Accepted in March 2007. To appear.

DeLine, Robert, and Manuel Fähndrich. 2001.
Enforcing high-level protocols in low-level software.
In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 59–69. Snowbird, UT:Acm Press.

Denning, Dorothy E. 1976.
A lattice model of secure information flow.
Communications of the ACM 19(5):236–243.

Denning, Dorothy E., and Peter J. Denning. 1977.
Certification of Programs for Secure Information Flow.
Communications of the ACM 20(7):504–513.

Dijkstra, Edgar. W. 1968.
Go to statement considered harmful.
Communications of the ACM 11(3):147–148.

Dreyer, Derek. 2005.
Recursive type generativity.
In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, 41–53. Tallinn, Estonia:Acm Press.

Fluet, Matthew, and Riccardo Pucella. 2006.
Phantom types and subtyping.
The Journal of Functional Programming 16(6):751–791.

Gallier, Jean H. 1990.
On Girard's "Candidats de Reductibilité".
Logic and Computer Science 31:123–203.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1994.
Design Patterns: Elements of Reusable Object-Oriented Software.

Reading, MA:Addison-Wesley.

Gentzen, Gerhard. 1935.
Untersuchungen über das Logische Schliessen.
Mathematische Zeitschrift 39:176–210 and 405–431.
English translation in (Gentzen 1969), pages 68–131.

———. 1969.
The collected papers of Gerhard Gentzen.
Studies in Logic and the Foundations of Mathematics, North-Holland.
Edited by M. E. Szabo.

Girard, Jean-Yves. 1972.
Interprétation functionelle et Élimination des coupures dans l'arithmétique d'order supérieure.
Thèse de doctorat, Université Paris VII.

Golubovsky, Dimitry, Neil Mitchell, and Matthew Naylor. 2007.
Yhc.Core - from Haskell to Core.
The Monad.Reader (7):45–61.

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. 2005.
The Java Language Specification.
3rd edition. Addison-Wesley.

Govereau, Paul. 2005.
Type generativity in higher-order module systems.
Technical Report TR-05-05, Harvard Computer Science.

Grossman, Dan, Greg Morrisett, and Steve Zdancewic. 2000.
Syntactic type abstraction.
Transactions on Programming Languages and Systems 22(6):1037–1080.

Harper, Robert, Furio Honsell, and Gordon Plotkin. 1993.
A framework for defining logics.
Journal of the ACM 40(1):143–184.

Harper, Robert, and Mark Lillibridge. 1994.
A type-theoretic approach to higher-order modules with sharing.
In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 123–137. Portland, OR:ACM Press.

Harper, Robert, and Greg Morrisett. 1995.
Compiling polymorphism using intensional type analysis.
In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 130–141. San Francisco, California:ACM Press.

Hasegawa, Masahito. 2005.
Relational parametricity and control.
In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, 72–81. Chicago, IL:Ieee Computer Society.

Heintz, Nevin C., and Jon G. Riecke. 1998.
The SLam calculus: programming with secrecy and integrity.
In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 365–377. San Diego, CA:Acm Press.

Hinze, Ralf. 2000.
Polytypic values possess polykinded types.
In *Proceedings of the 5th International Conference on Mathematics of Program Construction*, edited by Roland Backhouse and Jose Nuno Oliveira, volume 1837 of *Lecture Notes in Computer Science*, 2–27. Ponte De Lima, Portugal:Springer-Verlag.

Hinze, Ralf, and Andres Löh. 2006.
"Scrap your boilerplate" revolutions.
In *Proceedings of 8th International Conference on Mathematics of Program construction*, edited by Uustalu Tarmo, volume 4014 of *Lecture Notes in Computer Science*. Kuressaare, Estonia:Springer-Verlag.

Hinze, Ralf, Andres Löh, and Bruno C.d.S. Oliveira. 2006.
"Scrap your boilerplate" reloaded.
In *Proceedings of the 8th International Symposium on Functional and Logic Programming*, edited by Phil Waldler and Masami Hagiya, volume 3945 of *Lecture Notes in Computer Science*, 24–26. Fuji Susono, Japan:Springer-Verlag.

James H. Morris, Jr. 1973.
Types are not sets.
In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 120–124. Boston, MA:Acm Press.

Jansson, Patrik, and Johan Jeuring. 1998.
Functional pearl: Polytypic unification.
Journal of Functional Programming 8(5):527–536.

Jeuring, Johan, and Paul Hagg. 2002.
Generic programming for XML tools.
Technical Report UU-CS-2002-023, Utrecht University.

Johann, Patricia. 2002.
A generalization of short-cut fusion and its correctness proof.
Higher Order Symbolic Computation 15(4):273–300.

Kernighan, Brian W., and Dennis M. Ritchie. 1977.
The M4 Macro Processor.
Technical Report, Bell Laboratories, Murray Hill, New Jersey 07974.

Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. 2001.
An overview of AspectJ.
In *Proceedings of the 15th European Conference on Object-Oriented Programming*, edited by Jørgen Lindskov Knudsen, volume 2072 of *Lecture Notes in Computer Science*, 327–353. Budapest, Hungary:Springer-Verlag.

Kiczales, Gregor, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997.
Aspect-oriented programming.
In *Proceedings of the 11th European Conference on Object-Oriented programming*, edited by Mehmet Akşit and Satoshi Matsuoka, volume 1241 of *Lecture Notes in Computer Science*, 220–242. Jyväskylä, Finland:Springer-Verlag.

Krivine, Jean-Louis. 2001.
Typed lambda-calculus in classical Zermelo-Fraenkel set theory.
Archive of Mathematical Logic 40(3):189–205.

Kučan, Jakov. 2007.
Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS transform and "Free Theorems".
Ph.D. thesis, Massachusetts Institute of Technology.

Läfer, Konstantin, and Martin Odersky. 1994.
Polymorphic type inference and abstract data types.
ACM Transactions on Programming Languages and Systems 16(5):1411–1430.

Lämmel, Ralf, and Simon Peyton Jones. 2003.
Scrap your boilerplate: a practical design pattern for generic programming.
In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, 26–37. New Orleans, LA:Acm Press.

———. 2004.
Scrap more boilerplate: reflection, zips, and generalised casts.
In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, 244–255. Snow Bird, UT:Acm Press.

———. 2005.
Scrap your boilerplate with class: extensible generic functions.
In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, 204–215. Tallinn, Estonia:Acm Press.

Lamport, Leslie. 1977.
Proving the correctness of multiprocess programs.
IEEE Transactions on Software Engineering 3(2):125–143.

Lee, Daniel K., Karl Crary, and Robert Harper. 2007.
Towards a mechanized metatheory of Standard ML.
In *Proceedings of the 34th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 173–184. Nice, France:Acm Press.

Leifer, James J., Gilles Peskine, Peter Sewell, and Keith Wansbrough. 2003.
Global abstraction-safe marshalling with hash types.
In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, 87–98. Uppsala, Sweden:Acm Press.

Leroy, Xavier, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2000.
The Objective Caml system: Documentation and user's manual.
Available from `http://caml.inria.fr/`.

MacQueen, David, Gordon Plotkin, and Ravi Sethi. 1984.
An ideal model for recursive polymorphic types.
In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 165–174. Salt Lake City, UT:Acm Press.

McLean, John. 1994.
A general theory of composition for trace sets closed under selective interleaving functions.
In *Proceedings of the 1994 IEEE symposium on Security and Privacy*, 79. Oakland, CA:Ieee Computer Society.

Mellès, Paul-Andre, and Jérôme Vouillon. 2005.
Recursive polymorphic types and parametricity in an operational framework.
In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, 82–91. Chicago, IL:Ieee Computer Society.

Miller, Dale. 1991.
A logic programming language with lambda-abstraction, function variables, and simple unification.
In *Proceedings of the Second International Workshop on Extensions of Logic Programming*, edited by Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, volume 475 of *Lecture Notes in Computer Science*, 253–281. Tübingen, Germany:Springer-Verlag.

Milner, Robin, Mads Tofte, Robert Harper, and Dave MacQueen. 1997.
The Definition of Standard ML (revised).
Cambridge, MA:MIT Press.

Mitchell, Neil, and Colin Runciman. 2007.
Uniform boilerplate and list processing.

In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, 49–60. Freiburg, Germany:Acm Press.

Morrisett, Greg. 1995.
Compiling with types.
Ph.D. thesis, Carnegie Mellon University.
Tech Report Cmu-cs-95-226.

Mycroft, Alan. 1984.
Polymorphic type schemes and recursive definitions.
In *Proceedings of the 6th Colloquium on International Symposium on Programming*, edited by Manfred Paul and Bernard Robinet, volume 167 of *Lecture Notes in Computer Science*, 217–228. Toulouse, France:Springer-Verlag.

Myers, Andrew C., and Barbara Liskov. 2000.
Protecting privacy using the decentralized label model.
ACM Transactions on Software Engineering and Methodology 9(4):410–442.

Odersky, Martin. 2007.
Scala language specification version 2.6.
Draft available at `http://www.scala-lang.org/`.

Palsberg, Jens, and C. Barry Jay. 1998.
The essence of the visitor pattern.
In *Proceedings of the 22nd International Computer Software and Applications Conference*, 9–15. Vienna, Austria:Ieee Computer Society.

Parigot, Michel. 1992.
λ∘-calculus: An algorithmic interpretation of classical natural deduction.
In *International Conference on logic Programming and Automated Reasoning*, edited by Andrei Voronkov, volume 624 of *Lecture Notes in Computer Science*, 190–201. St. Petersburg, Russia:Springer-Verlag.

Parnas, D. 1972.
On the criteria to be used in decomposing systems into modules.
Communications of the ACM 15(12):1053–1058.

Peyton Jones, Simon. 2003.
Haskell 98 language and libraries: The revised report.
Cambridge, UK:Cambridge University Press.

Peyton Jones, Simon, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993.
The Glasgow Haskell compiler: a technical overview.
In *Proceedings of Joint Framework for Information Technology Technical Conference*, 249–257. Keele, UK.

Peyton Jones, Simon, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields.
Practical type inference for arbitrary-rank types.
Journal of Functional Programming To appear.

Peyton Jones, Simon, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006.
Simple unification-based type inference for GADTs.
In *Proceedings of the eleventh acm sigplan international conference on functional programming*, 50–61. Portland, OR:Acm Press.

Pfenning, Frank, and Conal Elliott. 1988.
Higher-order abstract syntax.
In *Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation*. Atlanta, GA:Acm Press.

Pfenning, Frank, and Carsten Schürmann. 1999.
System description: Twelf — a metalogical framework for deductive systems.
In *Proceedings of the 16th international conference on automated deduction*, edited by Harald Ganzinger, volume 1632 of *Lecture Notes in Computer Science*, 202–206. Trento, Italy:Springer-Verlag.

Pierce, Benjamin C., and David N. Turner. 2000.
Local type inference.
ACM Transactions on Programming Languages and Systems 22(1):1–44.

Pitts, Andrew. 2005.
Typed operational reasoning.
In *Advanced Topics in Types and Programming Languages*, edited by Benjamin C. Pierce, 245–289. MIT Press.

Pitts, Andrew, and Ian Stark. 1998.
Operational reasoning for functions with local state.
In *Higher Order Operational Techniques in Semantics*, edited by Andrew Gordon and Andrew Pitts, 227–273. Publications of the Newton Institute, Cambridge University Press.

Pottier, François, and Sylvain Conchon. 2000.
Information flow inference for free.
In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 46–57. Montreal, Canada:Acm Press.

Pottier, François, and Vincent Simonet. 2003.
Information flow inference for ML.
ACM Transactions on Programming Languages and Systems 25(1):117–158.

Ramsey, Norman. 1999.
Noweb – A Simple, Extensible Tool for Literate Programming.
Available from `http://www.eecs.harvard.edu/~nr/noweb/`.

Rémy, Didier. 1990.
Algèbres touffues. application au typage polymorphe des objets enregistrements dans les langages fonctionnels.
Thèse de doctorat, Université de Paris 7.

Reynolds, John C. 1974.
Towards a theory of type structure.
In *Programming Symposium, Proceedings Colloque sur la Programmation*, edited by Bernard Robinet, volume 19 of *Lecture Notes in Computer Science*, 408–423. Paris, France:Springer-Verlag.

———. 1983.
Types, abstraction, and parametric polymorphism.
In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*, edited by R. E. A. Mason, 513–523. Paris, France:Elsevier Science Publishers.

Rossberg, Andreas. 2003.
Generativity and dynamic opacity for abstract types.
In *Proceedings of the 5th international ACM SIGPLAN conference on Principles and practice of declarative programming*, edited by Dale Miller. Uppsala, Sweden:Acm Press.

Schneider, Fred B. 2000.
Enforceable security policies.
ACM Transactions on Information and System Security 3(1):30–50.

Schürmann, Carsten, and Jeffrey Sarnat.
Towards a judgmental reconstruction of logical relations proofs.
Available from `http://cs-www.cs.yale.edu/homes/sarnat/twelflist/twelflist-lr.pdf`.

Shinwell, Mark R., and Andrew M. Pitts. 2005.
Fresh Objective Caml User Manual.
Cambridge University Computer Laboratory. Available from `http://www.fresh-ocaml.org/`.

Simonet, Vincent. 2003.
Flow caml in a nutshell.
In *Proceedings of the first APPSEM-II workshop*, edited by Graham Hutton, 152–165. Nottingham, UK.

Sitaram, Dorai, and Matthias Felleisen. 1990.
Control delimiters and their hierarchies.
Lisp and Symbolic Computation 3(1):67–99.

Sperber, Michael, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. 2007.
Revised^6 report on the algorithmic language Scheme.
Robert Bruce Findler and Jacob Matthews authored the formal semantics. Available from `http://www.r6rs.org/`.

Steffen, Martin. 1998.
Polarized higher-order subtyping.
Ph.D. thesis, Technische Fakultät, Universität Erlangen.

Stirling, Colin. 2006.
A game-theoretic approach to deciding higher-order matching.
In *Proceedings (Part II) of the 33rd International Colloquium on Automata, Languages and Programming*, volume 4052 of *Lecture Notes in Computer Science*. Venice, Italy:Springer-Verlag.

Stroustrup, Bjarne. 2000.
The C++ Programming Language.
3rd edition. Reading, MA:Addison-Wesley.

Sumii, Eijiro, and Benjamin C. Pierce. 2003.
Logical relations for encryption.
Journal of Computer Security 11(4):521–554.

———. 2005.
A bisimulation for type abstraction and recursion.
In *Popl '05: Proceedings of the 32nd acm sigplan-sigact symposium on principles of programming languages*, 63–74. New York, NY, USA:Acm Press.

———. 2007.
A bisimulation for dynamic sealing.
Theoretical Computer Science 375(1–3):169–192.

Sun Microsystems. 1997.
JavaBeans API specification 1.01.
`http://java.sun.com/products/javabeans/`.

Swamy, Nikhil, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006.
Safe manual memory management in Cyclone.
Science of Computer Programming 62(2):122–144.

Syme, Don, and James Margetson. 2006.
F# manual.
Available from `http://research.microsoft.com/fsharp/`.

Ecma. 2006.
ECMA-334: C# language specification.
4th edition. Geneva, Switzerland:Ecma International.
This standard is also approved as Iso/iec 23270:2006.

Tse, Stephen, and Steve Zdancewic. 2004a.
Run-time Principals in Information-flow Type Systems.
In *IEEE 2004 Symposium on Security and Privacy*. Oakland, CA:IEEE Computer Society Press.

———. 2004b.
Translating dependency into parametricity.
In *Proceedings of the 9th ACM SIGPLAN international conference on Functional programming*. Snowbird, Utah:Acm Press.

———. 2005.
Designing a security-typed language with certificate-based declassification.
In *Proceedings of the 14th european symposium on programming*, edited by Shmuel Sagiv, volume 3444 of *Lecture Notes in Computer Science*. Edinburgh, UK:Springer-Verlag.

Turon, Aaron. 2007.
The SML/NJ Language Processing Tools: User guide.
For sml/nj version 110.65.

Vestin, Møns. 1997.
Genetic algorithms in haskell with polytypic programming.
Master's thesis, Göteborg University.

Vitek, Jan, and Boris Bokowski. 1999.
Confined types.
In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 82–96. Denver, CO:Acm Press.

Volpano, Dennis, Geoffrey Smith, and Cynthia Irvine. 1996.
A sound type system for secure flow analysis.
Journal of Computer Security 4(3):167–187.

Vytiniotis, Dimitrios, Geoffrey Washburn, and Stephanie Weirich. 2005.
An open and shut typecase.
In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, 13–24. ACM SIGPLAN, Longbeach, CA:Acm Press.

Vytiniotis, Dimitrios, and Stephanie Weirich. 2007a.
Free theorems and runtime type representations.
In *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics*, volume 173 of *Electronic Notes in Theoretical Computer Science*, 357–373. New Orleans, LA:Elsevier Science Publishers.

———. 2007b.
Type-safe cast does no harm.
Draft available from `http://www.cis.upenn.edu/~sweirich/papers/popl08-parametricity.pdf`.

Wadler, Philip. 1989.
Theorems for free!
In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, 347–359. London, UK:Acm Press.

Wand, Mitchell. 1987.
Complete type inference for simple objects.
In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, edited by David Gries, 37–44. Ithaca, NY:Ieee Computer Society Press.

———. 1988.
Corrigendum: Complete type inference for simple objects.
In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, edited by Yuri Gurevich, 132. Edinburgh, UK:Ieee Computer Society Press.

Washburn, Geoffrey, and Stephanie Weirich. 2005.
Generalizing parametricity using information flow.
In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, 62–71. Chicago, IL:IEEE Computer Society.

Weirich, Stephanie. 2006.
RepLib: a library for derivable type classes.
In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, 1–12. Portland, OR:Acm Press.

de Wit, Jan. 2002.
A technical overview of Generic Haskell.
Master's thesis, Utrecht University.
Technical report Inf-scr-02-03.

Wright, Andrew K., and Matthias Felleisen. 1994.
A syntactic approach to type soundness.
Information and Computation 115(1):38–94.

Xi, Hongwei, Chiyan Chen, and Gang Chen. 2003.
Guarded recursive datatype constructors.
In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 224–235. New Orleans, LA:Acm Press.

Zdancewic, Stephan. 2002.
Programming languages for information security.
Ph.D. thesis, Cornell University.

Zheng, Lantian, and Andrew C. Myers. 2004.
Dynamic security labels and noninterference.
In *Formal Aspects in Security and Trust*, volume 173 of *IFIP International Federation for Information Processing*, 27–40. Toulouse, France:Springer-Verlag.

Zwanenburg, Jan. 1999.
Pure type systems with subtyping.

In *Proceedings of the 4th International Conference Typed Lambda Calculi and Applications*, edited by Jean-Yves Girard, volume 1581 of *Lecture Notes in Computer Science*, 381–396. L'Aquila, Italy:Springer-Verlag.

Colophon

> Ah, typography! Way number one hundred twenty-one to avoid graduating!
>
> Peter Sewell (2005)

This document was prepared using the LaTeX typesetting system created by Leslie Lamport, with Peter Wilson's Memoir class. I used Chung-chieh Shan's (單中杰) bibliography style "McBride" with some modifications. The document was processed using pdfTeX's microtypography extensions implemented by Hàn Thế Thành. The body text is set at 10pt. The serif typefaces are from the Warnock® Pro Opticals family, designed by Robert Slimbach of Adobe Systems Incorporated. The sans-serif typefaces are from the Cronos® Pro Opticals family, also designed by Robert Slimbach. The LaTeX infrastructure for using these two typefaces was generated by the tool otftofd, designed and written by myself. The monospace typeface used for typesetting source code is Deja Vu Sans Mono, which has been developed by many contributors. The serif and calligraphic mathematical typefaces are from ams Euler, designed by Hermann Zapf, with adjustments to the sidebearings and kerning. The typeface for mathematical symbols is Gentzen Symbol, designed by myself.